Spring 2018

Approaches to Shared State in Concurrent Programs

Sidharth Mishra
San Jose State University

Approaches to Shared State in Concurrent Programs

A Project Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Sidharth Mishra

May 2018

The Designated Project Committee Approves the Project Titled

Approaches to Shared State in Concurrent Programs

By

Sidharth Mishra

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2018

Dr. Jon Pearce	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science

Abstract

Approaches to Shared State in Concurrent Programs

By Sidharth Mishra

We are in the multicore machine era, but our programs have yet to utilize the increased computing power offered by these machines. At present, lock-based multithreaded programming is the most common programming model used for writing concurrent programs. However, due to the nuances of *shared state* (and memory) in multithreaded programs and the cognitive load introduced due to locks, concurrent programming remains difficult. One way to deal with shared state in concurrent programs is to get rid of it altogether and use message passing. The other way would be to isolate shared state and store it in a *state store*, making it the "*single source of truth*". This paper explores the problems with lock-based multithreaded programming and discusses approaches for handling shared state in concurrent programs. We introduce a novel pattern language called *Quarantined Software Transactional Memory* (QSTM) and use it to solve the nuances of shared state in concurrent programs. Subsequently, we introduce the *monad pattern language* for making implicit side-effects in a program explicit and discuss its incorporation into the QSTM pattern. Finally, we present a comparison between the QSTM pattern and *Redux* — a popular JavaScript-based *state store*.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Jon Pearce for his endless support and guidance throughout the course of this project. I would also like to thank my committee members Dr. Thomas Austin and Dr. Chris Pollett for their valuable time and feedback.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1 INTRODUCTION

In this paper, we propose a pattern language to deal with the nuances of shared state (and memory) by isolating it in a variation of the *Software Transactional Memory* (STM) [1]. We call the pattern *Quarantined Software Transactional Memory* (QSTM); we provide two example implementations of the pattern using *Java* and *Go* [2] programming languages. Furthermore, we also propose a pattern language for *Monads* and incorporate it into our QSTM pattern for protecting against *hard-to-rollback-actions* while writing *transactional actions*.

According to the comprehensive study by Lu et al. in [3], most of the non-deadlock bugs in concurrent programs are *atomicity-violation* and *order-violation* bugs. They claim that by using a simple *transactional memory* (TM) implementation (which only guarantees *atomicity* and *isolation* when executing operations), we can avoid **39%** of their observed concurrency bugs. Since our QSTM pattern also provides the same guarantees when executing the operations, if applied to the scenarios discussed by Lu et al., we too can avoid those same concurrency bugs – transitively. However, unlike the simple TM implementation, our QSTM does provide semantics to specify execution order intentions; it helps in avoiding additional **19%** of the concurrency bugs that are caused by violating the programmer's order intentions – these bugs are difficult to address using a simple TM implementation.

Furthermore, our *monadic* QSTM provides semantics to protect against *hard-to-rollback-actions*. This enables it to help in avoiding some of their examined bugs (**42%**) which cannot be addressed by the simple TM and normal QSTM implementations.

1.1 MOTIVATION

We are in the age of multicore machines, but we are still behind when talking about software that utilizes these machines. As Sutter and Larus point out in [4], although we have several programming models designed for concurrent programs, these models are only applicable to specific scenarios. Moreover, it is quite difficult to predict which programming model would be a better fit for a particular problem and combining several of these models is problematic.

Threads represent the fundamental concurrency model supported by most modern programming languages and operating systems. However, threads are non-deterministic [4, 5]. Often, the concurrent computations implemented using threads differ and their data accesses are unpredictable, requiring explicit synchronization via locks, monitors, etc. Because threads share memory, the unorganized shared memory accesses with no form of synchronization lead to data races.

Synchronization is a necessity for pruning away the unpredictability of shared memory access and preventing data races. But, synchronization is hard. The simplest synchronization mechanism available to programmers is the lock [4], although pretty simple, it introduces a cognitive overhead, and a new class of problems: deadlock, livelock, etc.

Furthermore, lock-based programs are not composable like normal object-oriented programs and functions. It is difficult to call into external lock-based libraries without reviewing their implementations because they may lead to deadlocks. Additionally, the relationship between the data and the lock that is associated with it is not explicitly specified and is entirely dependent on programmer discipline. Lee in [5] demonstrates the increase in program complexity when translating a sequential *Java* program into a concurrent one and conjectures

that most multithreaded applications are full of concurrency bugs that will show up as system failures as multicore machines become common.

In order to tackle the problem of unorganized shared state in multithreaded programs, we can take two approaches. The first approach would be to give up shared state altogether and move over to *message passing*. The *actor model* and *active object model* would be great examples of this style – *Akka* [6], an actor framework for the *Java Virtual Machine* (JVM) has been gaining popularity recently. The second approach would be to organize the unorganized shared memory (and state) in a single location and isolate it. *Software transactional memory* (STM) introduced by Shavit and Touitou [1] does just that. The STM organizes the shared memory in one place and isolates it from the rest of the program. It allows modification of this shared memory only through database-like transactions. These transactions are *atomic* and *serializable* [1, 7, 8]. Functional languages like *Clojure* [9] and *Haskell* [10] have embraced the STM as the way to implement *mutable shared state*.

1.2 OVERVIEW

This paper focuses on STM as a design pattern and proposes a pattern language for managing shared state in concurrent programs using a variation called the *Quarantined Software Transactional Memory* (QSTM). This section describes the flow of content in this paper. First, in Section 2, we will discuss the problems with lock-based programs and possible alternatives to the lock-based programming model. Section 3 will provide a brief overview of the message passing approach to achieve concurrency.

Second, we will move to the *isolating shared state* approach in Section 4 — this will also serve as a gentle introduction to the *QSTM pattern language*. Section 4.2 will take a deeper dive

into the *QSTM pattern language*, and we will take a look at few possible implementations for the pattern in Section 4.3 and Section 4.4.

Then, we will look into the *Monad pattern* in Section 5 and discuss how it fits well into the *QSTM pattern* in Section 5.3. Finally, Section 6 will provide an overview of *Redux* [11] and its comparison with our *QSTM pattern.*

2 PROBLEMS WITH LOCK-BASED PROGRAMS

The multithreaded programming model is difficult. The literature surveyed for this paper [4, 5] unanimously declare that the current thread-and-lock-based programming model is not good enough for designing large-scale concurrent programs. We will take a simple bank account simulation as an example to demonstrate the increase in code complexity as we introduce threads and locks (synchronization tools). Listing 1 shows the *Java* class definition of a domain object, *Account*, when executing in sequential mode.

The class has fields: *ID, name,* and *balance*; it also has methods: *deposit, withdraw,* and *transfer.* This class definition is good for only the sequential execution scenario. The moment we introduce concurrency the source code gains complexity as seen in Listing 2.

Threads execute non-deterministically, and since they share memory, explicit synchronization is needed to crop out the non-determinism. Locks are the popular choice for achieving the desired synchronization. However, lock-based programs (libraries) are not composable [4] (or modular). Importing and using two or more lock-based libraries written by different authors in a concurrent program is not a trivial matter. Proper care needs to be taken in-order to avoid deadlocks. Moreover, calling into external lock-based libraries without looking at their definitions can often lead to deadlocks [4, 7]. With modular programming becoming the industry standard, the inability to compose two or more pieces of lock-based concurrent programs is a liability. Moreover, locks are low-level programming constructs and often programming languages do not have built-in standards to provide explicit information about the lock [7, 5, 4]. For example, in *Java*, there is no way to explicitly express information about the resource a lock is protecting.

```
/**
 * The bank account is the domain object.
 */
public class Account {

  /**
   * The unique identifier of the account.
   * Generated from UUID 4.
   */
  private @Getter String ID;

  /**
   * The name of the account.
   */
  private @Getter String name;

  /**
   * The balance of the account. For simplicity's sake,
   * let’s assume the balance is an integer.
   */
  private @Getter int balance;

  /**
   * Creates a new bank account.
   *
   * @param name
   *          The name of the account.
   * @param balance
   *          The initial balance of the account.
   */
  public Account(String name, int balance) {
    this.ID = UUID.randomUUID().toString();
    this.name = name;
    this.balance = balance;
  }

  /**
   * Deposit adds the amount to the account's balance.
   * Note: This is a behavior that mutates the state of the account.
   *
   * @param amount
   *          The amount to deposit.
   */
  public void deposit(Integer amount) {
    if (Objects.isNull(amount)) return;
    this.balance = this.balance + amount;
  }

  // -------------- ACCOUNT CLASS DEFINITION CONTINUED -------------
```

```
  // -------------- ACCOUNT CLASS DEFINITION CONTINUES ----------

/**
 * Withdraw reduces the balance if it is enough, else does nothing.
 * Note: This is a behavior that mutates the state of the account.
 *
 * @param amount
 *          The amount to withdraw from the account.
 *
 * @throws Exception
 */
  public void withdraw(Integer amount) throws Exception {
    if (amount > this.balance || Objects.isNull(amount))
        throw new Exception("Balance too low to withdraw");
    this.deposit(-1 * amount);
  }

  /**
   * Transfers the desired amount from this account to the destination account.
   *
   * @param dest
   *          The destination account.
   * @param amount
   *          The desired amount.
   */
  public void transfer(Account dest, Integer amount) {
    try {
      this.withdraw(amount);
      dest.deposit(amount);
    } catch (Exception e) {
      System.err.println("Failed to transfer, insufficient funds");
    }
  }
}

// -------------- ACCOUNT CLASS DEFINITION ENDS ----------
```

Listing 1 Sequential Account class definition in Java.

Just by looking at the definition of the class *Account* in Listing 2, it is not clear what resource the lock is protecting. Since there are no language features to make this explicit, it has to be done through documentation, grouping, or sometimes enterprise specific policies or development guidelines [4, 7]. For example, the *Go* [2] programming guidelines/conventions in [12] specify that locks be grouped together with the data they protect.

```
/**
 * The bank account is the domain object.
 */
public class Account {

  /**
   * The unique identifier of the account.
   * Generated from UUID 4.
   */
  private @Getter String ID;

  /**
   * The name of the account.
   */
  private @Getter String name;

  /**
   * The balance of the account. For simplicity's sake,
   * let’s assume the balance is an integer.
   */
  private @Getter int balance;

  /**
   * The lock for this account.
   */
  private ReentrantLock lock;

  /**
   * Creates a new bank account.
   *
   * @param name
   *          The name of the account.
   * @param balance
   *          The initial balance of the account.
   */
  public Account(String name, int balance) {
    this.ID = UUID.randomUUID().toString();
    this.name = name;
    this.balance = balance;
  }

// ------------- CONCURRENT ACCOUNT CLASS DEFINITION CONTINUED -----------
```

```
// ------------- CONCURRENT ACCOUNT CLASS DEFINITION CONTINUES --------------

  /**
   * Deposit adds the amount to the account's balance.
   *
   * Note: This is a behavior that mutates the state of the account.
   *
   *          The amount to deposit.
   */
  public void deposit(Integer amount) {

    try {

      this.lock.lock();

      if (Objects.isNull(amount)) return;

      this.balance = this.balance + amount;

    } finally {

      this.lock.unlock();
    }
  }

  /**
   * Withdraw reduces the balance if it is enough, else does nothing.
   *
   * Note: This is a behavior that mutates the state of the account.
   *
   * @param amount
   *          The amount to withdraw from the account.
   *
   * @throws Exception
   */
  public void withdraw(Integer amount) throws Exception {

    try {

      this.lock.lock();

      if (amount > this.balance || Objects.isNull(amount))
          throw new Exception("Balance too low to withdraw");

      // No longer easy to call deposit() without modifying
      // the definition of deposit().
      //
      this.balance = this.balance - amount;

    } finally {

// ------------- CONCURRENT ACCOUNT CLASS DEFINITION CONTINUED ---------------
```

```
// ------------- CONCURRENT ACCOUNT CLASS DEFINITION CONTINUES ----------

      this.lock.unlock();
    }
  }

  /**
   * Transfers the desired amount from this account to the destination account.
   *
   * Note: This is a behavior that mutates the state of the account.
   *
   * @param dest
   *          The destination account.
   * @param amount
   *          The desired amount.
   */
  public void transfer(Account dest, Integer amount) {

    try {

      // taking locks on both the accounts
      //
      this.lock.lock();
      dest.lock.lock();

      // the transfer operation is atomic
      //
      this.withdraw(amount);
      dest.deposit(amount);

    } catch (Exception e) {

      System.err.println("Failed to transfer, insufficient funds");

    } finally {

      // releasing locks in the reverse order - unlock order is important!
      //
      dest.lock.unlock();
      this.lock.unlock();
    }
  }

 . . .

}

// ------------- CONCURRENT ACCOUNT CLASS DEFINITION ENDS ---------------
```

```
// ---------------- SIMULATION DRIVER CODE -START - -----------

  /**
   * @param args
   */
  public static void main(String[] args) {

    // Create two accounts
    //
    Account acc1 = new Account("account1", 100);
    Account acc2 = new Account("account2", 200);

    // print the strings for acc1 and acc2, cheap debugging.
    //
    logger.info("Account 1:\n" + acc1.toString());
    logger.info("Account 2:\n" + acc2.toString());

    // Create a threadPool having 4 threads to simulate multiple threads
    // requesting changes on the accounts.
    //
    ExecutorService threadPool = Executors.newFixedThreadPool(4);

    // perform the actions on the accounts
    // deposit 50 into acc1: acc1 = 150
    // deposit 300 into acc2: acc2 = 500
    // transfer 35 from acc1 to acc2: acc1 = 150 - 35 = 115, acc2 = 500 + 35 = 535
    // transfer 50 from acc2 to acc1: acc1 = 115 + 50 = 165, acc2 = 535 - 50 = 485
    //
    // Finally, acc1 = 165, acc2 = 485 (Test for consistency of the operations)
    //
    // Note: all these actions happen to be running on separate threads under the hood.
    //
    threadPool.submit(() -> acc1.deposit(50));
    threadPool.submit(() -> acc2.deposit(300));
    threadPool.submit(() -> acc1.transfer(acc2, 35));
    threadPool.submit(() -> acc2.transfer(acc1, 50));

    shutdown(threadPool);

    // print the strings for acc1 and acc2, cheap debugging.
    //
    logger.info("Account 1:\n" + acc1.toString());
    logger.info("Account 2:\n" + acc2.toString());
  }

// ---------------- SIMULATION DRIVER CODE -END -  -----------
```

Listing 2 Lock based concurrent definition for Account class.

Furthermore, locks bring in additional cognitive load for the programmer. Taking fewer locks may lead to *data races*. However, by taking many locks one risks a deadlock or degraded

performance. Since, there is no explicit way to know about the resources a lock is protecting, taking wrong locks, or taking the locks in the wrong order is commonplace — several errors arise due to bad locking practices. Moreover, it is difficult for a programmer to guarantee consistency of an application when dealing with locks.

The main cause of concern for the programmers in concurrent lock-based programs is shared state. One of the ways to deal with shared state would be to get rid of it completely. When we get rid of shared state, we move away from a shared memory model and enter a message passing model. *Active objects*, *actor model*, and multiprocessing are some of the best-known examples of this style of concurrent programming. Moreover, this model is highly scalable since the components do not share memory and can be moved over to other machines if needed – this translates well into distributed computing models. Section 3 provides an overview on *active objects* and *actor model*.

The second way to deal with shared state in concurrent programs would be to *isolate the shared state* at one place — in a *shared state store* — and have all the threads access this shared *state store* in a deterministic/constrained way. Section 4 introduces our QSTM pattern that isolates the shared state and stores it in a STM.

3 MESSAGE PASSING

In this programming model, we get rid of all shared state and instead try to achieve concurrency by sending and receiving computation and data as messages. The simplest way to visualize this would be through the *active object pattern* in Section 3.1. The *actor model* in Section 3.2 is an extension of the *active object pattern* and allows the building of highly scalable distributed and concurrent applications.

3.1 ACTIVE OBJECT

An active object is an object that has its own thread of control. In this pattern, we separate the object's method execution from its method invocation by introducing a message queue for each object. The method invocations on the object can be seen as incoming messages to the object which are then added to the object's message queue. The object then processes these messages one at a time, executing the code specific to the message received. Lavender and Schmidt in [13] state that this model of concurrent programming is generally well suited for producer/consumer and reader/writer applications.

Producer/Consumer applications are ones in which one part of the application acts as the emitter or producer of messages in response to specific events. The other part acts as the receiver or consumer of the messages. Upon receiving the message, the consumer part takes action. An example would be a file watcher application that keeps watching a file for changes. When the file changes, the producer emits a message that is then received by the consumer and action is taken in response. The action could be executing a specific script like formatting the source or validating the source, etc. Having both the producer and consumer implemented as active objects

allows for separation of concerns and better concurrency. The logic could also be extended to a multiple process architecture since there is no shared state or memory.

3.2 ACTOR MODEL

Actor Model is an extension of the *active object design pattern*. Each *actor* in the *actor model* is an *active object*. The *actor* sends and receives *messages* from other actors. They do not share state and communicate solely via messages. Each *actor* is responsible for its own state which is implemented as the actor's attributes. The actor's state is modified by its behaviors or operations in response to the messages received from other actors.

Similar to the *active object*, an *actor* has a *message queue* and runs in its own thread of control. When a *message* is passed to an *actor*, it gets added to the actor's *message queue*. The actor keeps polling the messages from its *message queue* one at a time and executes the behavior/operation corresponding to the polled message. Lavender and Schmidt in [13] and Agha in [14, 15, 16] sketch a possible actor model implementation using the *active object design pattern*.

Akka [6] is an enterprise grade actor model framework/toolkit designed for the JVM. *Akka's Actors Toolkit* is written in *Scala* [17] and borrows its syntax from *Erlang* [18]. It provides a higher-level abstraction for writing distributed and concurrent applications.

4 ISOLATING SHARED STATE

One way to tackle the nuances of shared state in a concurrent program is to *isolate* the *shared state* and keep it in a *shared state store*. This *state store* becomes the "single source of truth". Our *Quarantined Software Transactional Memory* (QSTM) pattern language focuses on this style of memory arrangement for concurrent programming. It is based on Shavit and Touitou's *Software Transactional Memory* (STM) [1], and is inspired by Jones' approach of constructing an STM in [7].

We will extend on the scenario introduced in Section 2 to showcase this novel pattern. In this simulation, our *domain object* called *Account* represents the bank account. It has certain read-only attributes: *accountName, ID, creationDate, holderName,* etc. Also, it has mutable attributes: *balance, lastUpdateDate,* etc. It also has operations: *deposit, withdraw, and transfer*.

The first step in isolating the shared state is to segregate the *identity* from the *state* of the domain object — segregate the *immutable* from *mutable*. Generally, most domain objects are big chunks of *identity* bundled with *state*. Moreover, the *operations* of the objects are basically modifying their *states*. The *identities* of the objects remain unchanged throughout the lifetime of the object — *identity* is immutable. Hence, it is safe to access the *identity* of the object from multiple threads. It is only the *state* that is the cause of concern.

In our case, we will split the *Account,* class into two classes: *AccountDetails,* and *AccountState*. The *AccountDetails* class, represents the identity of the account object. Similarly, the AccountState class represents the state of the domain object. The UML class diagram in Figure 1 shows how the domain object splits up into identity and state.

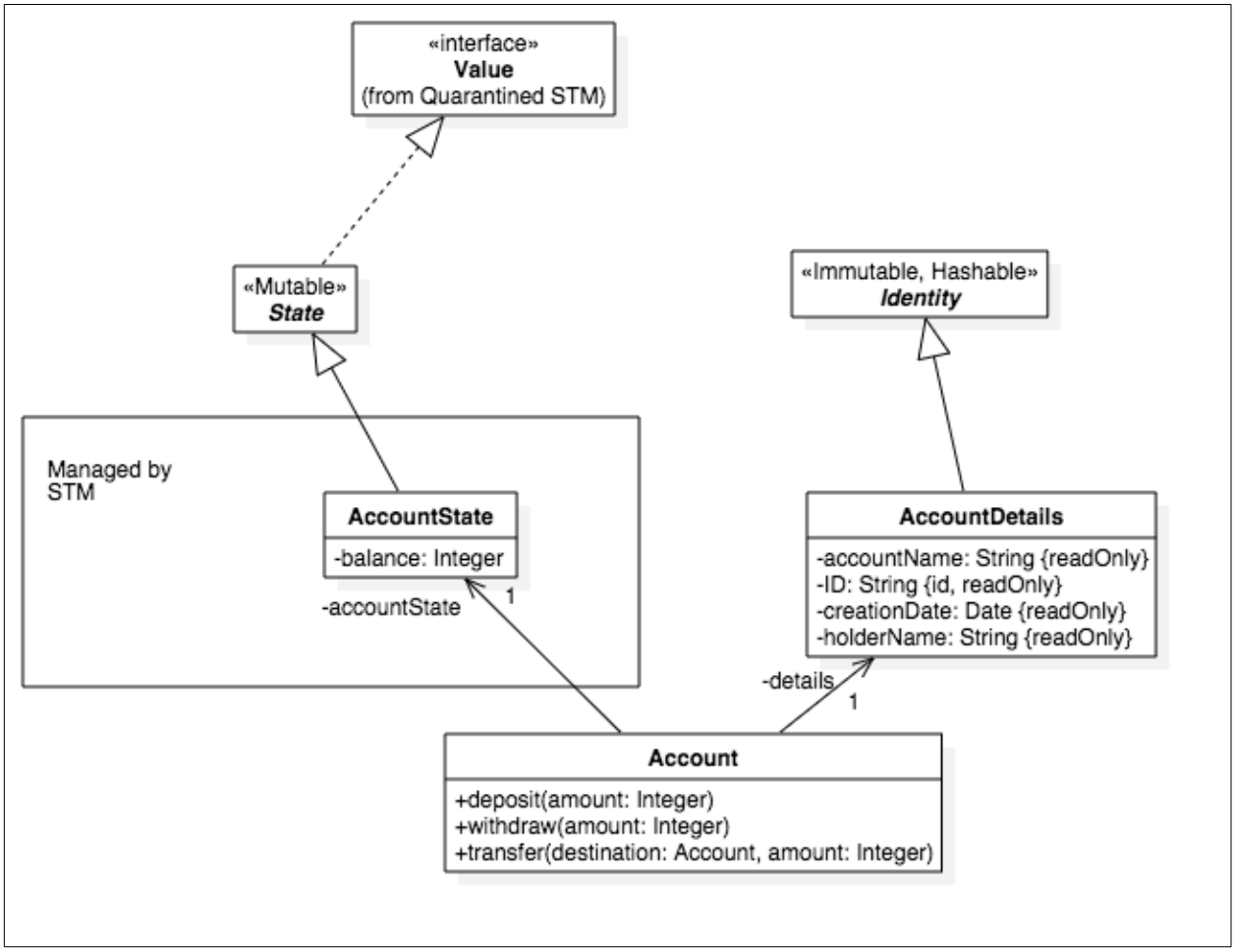

Figure 1 Splitting Account into identity and state.

Now, after we have segregated the *state* of the domain object, we need to isolate it in a container that does not allow direct modification. The container is the QSTM and the *state* being contained/managed by the QSTM can only be modified through special operations called *transactions*. We will look at the *Quarantined Software Transactional Memory* (QSTM) pattern language in detail in Section 4.2 but, for now, let us assume that anything that needs to be stored/managed in the QSTM needs to conform to the QSTM's *Value* interface. A *value* needs to be *cloneable* and *equitable* — more information on these requirements to follow in Section 4.2.

In order to satisfy the requirements of the QSTM, we create a domain specific abstract class called *State* that realizes the QSTM's *Value* interface. Similarly, we also create a domain specific abstract class called *Identity.* The *AccountDetails* class is the concrete implementation of the *Identity* class and *AccountState* is the concrete implementation for *State*. The class hierarchy is visualized in the UML diagram in Figure 1.

With the QSTM managing our shared state (which are *AccountState* instances) our domain object's definition changes from our sequential and lock-based definitions. The source code snippets in Listing 3, Listing 4, and Listing 5 hold the new definitions for the classes *AccountDetails*, *AccountState*, and *Account* respectively.

```
/**
 * The identity information about the account. It is immutable.
 */
public class AccountDetails extends Identity {

  /**
   * The unique identifier of the account. Generated from UUID 4.
   */
  private @Getter String ID;

  /**
   * The name of the account.
   */
  private @Getter String name;

  /**
   * The account details.
   *
   * @param name
   *          The name of the account.
   */
  public AccountDetails(String name) {
    this.ID = UUID.randomUUID().toString();
    this.name = name;
  }

}
```

Listing 3 AccountDetails class definition.

```
/**
 * The mutable part of the bank account. It holds the contents that represent
 * the state of the account.
 *
 * The state of an account changes throughout its lifetime, and when shared
 * across multiple threads is a cause for
 * concern.
 *
 */
public class AccountState extends State {

  /**
   * The balance of the account. For simplicity's sake, lets assume the balance is
   * an integer.
   */
  private @Getter int balance;

  /**
   * Creates a 0 balance account state.
   */
  public AccountState() { . . . }

  /**
   * Creates the account's state.
   *
   * @param balance
   *          The starting balance for the account.
   */
  public AccountState(int balance) { . . . }

  /*
   * (non-Javadoc)
   * @see stm.Value#makeCopy()
   */
  @Override
  public Value makeCopy() { . . . }

  /*
   * (non-Javadoc)
   * @see stm.Value#isEqual(stm.Value)
   */
  @Override
  public Boolean isEqual(Value v) { . . . }

  // -------- CLASS DEFINITION CONTINUED ------------
```

```
// ---------------- ACCOUNTSTATE CLASS DEFINITION CONTINUES ---------

/**
   * Deposit adds the amount to the account's balance.
   *
   * Note: This is a behavior that mutates the state of the account.
   *
   * @param amount
   *          The amount to deposit.
   */
  public void deposit(Integer amount) {
    if (Objects.isNull(amount)) return;
    this.balance = this.balance + amount;
  }

  /**
   * Withdraw reduces the balance if it is enough, else does nothing.
   *
   * Note: This is a behavior that mutates the state of the account.
   *
   * @param amount
   *          The amount to withdraw from the account.
   * @throws Exception
   */
  public void withdraw(Integer amount) throws Exception {
    if (amount > this.balance) throw new Exception("Balance too low to withdraw");
    this.deposit(-1 * amount);
  }

// -------- IRRELEVANT CODE FOR OTHER METHODS -------

}

// ------------- ACCOUNTSTATE CLASS DEFINITION ENDS -----------------
```

Listing 4 AccountState class definition.

The class definition of *AccountState* in Listing 4 shows the two methods: *makeCopy* and *isEqual*. These methods are needed for *AccountState* to conform to the QSTM's *Value* interface. These methods make the *AccountState* object *cloneable* and *equitable* — so that two states can be equated. The definition for *deposit* and *withdraw* methods remain unaltered from the sequential version for *Account* in Listing 1.

```
/**
 * The bank account is the domain object. We segregate its content into identity
 * and state. We let the STM manage the state. This ensures safe concurrency.
 */
public class Account {
  /**
   * The identity part of the account.
   */
  private @Getter AccountDetails details;

  /**
   * The state part of the account. It is maintained inside the STM.
   * So, we get the transactional variable that refers to the memory cell
   * where the account state is actually stored.
   */
  private @Getter TVar accountState;

  /**
   * The reference to the STM that will be managing the state of this account.
   */
  private @NonNull STM stm;

  /**
   * Creates a new bank account.
   *
   * @param accountName
   *          The name of the account.
   * @param initialBalance
   *          The initial balance in the account.
   */
  @Builder
  public Account(String accountName, Integer initialBalance, STM stm) {
    AccountDetails details = new AccountDetails(accountName);
    this.details = details;
    this.stm = stm;

    // storing the account's state in the STM for safe management
    //
    this.accountState = this.stm.newTVar(new AccountState(initialBalance));
  }

  //  ---------------------- ACCCOUNT CLASS DEFINITION CONTINUED -------
```

```
  // -------------- ACCOUNT CLASS DEFINITION CONTINUES ----------------------

  /**
   * Deposits the amount into this account.
   *
   * @param amount
   *          The amount to be deposited into the account.
   */
  @SuppressWarnings("unchecked")
  public void deposit(Integer amount) {
    // SUBMIT A JOB TO THE STM FOR UPDATING STATE.
  }

  /**
   * Withdraws the specified amount from this account.
   *
   * @param amount
   *          The amount to be withdrawn from the account.
   */
  @SuppressWarnings("unchecked")
  public void withdraw(Integer amount) {
    // SUBMIT A JOB TO THE STM FOR UPDATING STATE.
  }

  /**
   * Transfers the desired amount from this account to the destination account.
   *
   * @param destination
   *          The account to transfer to.
   * @param amt
   *          The amount to transfer.
   */
  @SuppressWarnings("unchecked")
  public void transfer(Account destination, Integer amt) {
    // SUBMIT A JOB TO THE STM FOR UPDATING THE STATES OF
    // THIS ACCOUNT AND THE DESTINATION ACCOUNT.
  }
}

// -------------- ACCOUNT CLASS DEFINITION ENDS ----------------------
```

Listing 5 Account class definition using QSTM pattern language.

As can be seen from the code in Listing 5, unlike in case of locks, we know exactly what resource is being managed by the STM. This is because any data being stored in the STM is referenced by a *TVar* (*transactional-variable*, more details in Section 4.2). This explicit

information reduces the cognitive load on the programmer because the source code itself becomes the standard and documentation.

Also, in order to modify the state being managed by the QSTM, we need to submit a job/action (more details in Section 4.2) — *transactional-action* — to the STM. Submitting jobs to a thread-pool can be viewed as an analogy to this scenario. The methods *deposit, withdraw,* and *transfer* in Listing 5 all submit *transactional-actions* to the STM to modify the account's state. The exact implementations of these methods will be discussed after the introduction of the QSTM in Section 4.2.

4.1 SOFTWARE TRANSACTIONAL MEMORY

Software Transactional Memory (STM) is a novel way for translating sequential code into concurrent code without having to deal with lower level synchronization tools: locks, semaphores, monitors, etc. As pointed out in [4], we are lacking tools that provide abstractions for concurrent programming. Hence, STM with its high-level interface and modularity becomes indispensable.

The STM was introduced by Shavit and Touitou in [1]. Henceforth, there have been several implementations of the STM: [19, 20, 21, 22]. Several functional languages: *Clojure* [9], *Haskell* [10] have already incorporated STM as standard library features. Jones in [7] explains a possible implementation strategy of the STM in *Haskell*. Our QSTM pattern language is based on the Shavit and Touitou's STM [1] and is inspired by Jones' implementation in [7].

4.2 QSTM

Quarantined Software Transactional Memory (QSTM) pattern language is a pattern language that makes use of a variation of the STM to manage shared state in concurrent programs.

4.2.1 QSTM Pattern Language

```
Name: Quarantined Software Transactional Memory (QSTM)
Context: You are building a concurrent application using the multithreaded programming
model.
Problem: Lock based multithreaded programming model is complicated when there is shared
state. The programmer needs better abstraction and modularity.
Solution: First, decouple identity and state of the domain object. Second, store the
state of the domain object in the QSTM's STM and let the STM manage it. Third, model the
domain object's operations to submit transactional actions to the STM rather than
modifying the state directly.
Consider these patterns next: Active Object, Actor Model, Communicating Sequential
Process (CSP).
```

Listing 6 QSTM pattern language.

The QSTM pattern language in Listing 6 makes use of the STM to manage shared state in concurrent programs. To start off, we split the domain object into *identity* and *state* — covered in Section 4. Then, we store the *state* in the STM and let the STM manage the *state* safely.

Following the convention of the layered design pattern in [23], the QSTM's STM implementation sits at the *infrastructure layer*. It works as a framework on which the *domain*

specific framework is built upon. The dependency is tightly coupled as we move from top to down in the *layered pattern* (see Figure 2), making the STM implementation in the infrastructure layer highly modular and reusable.

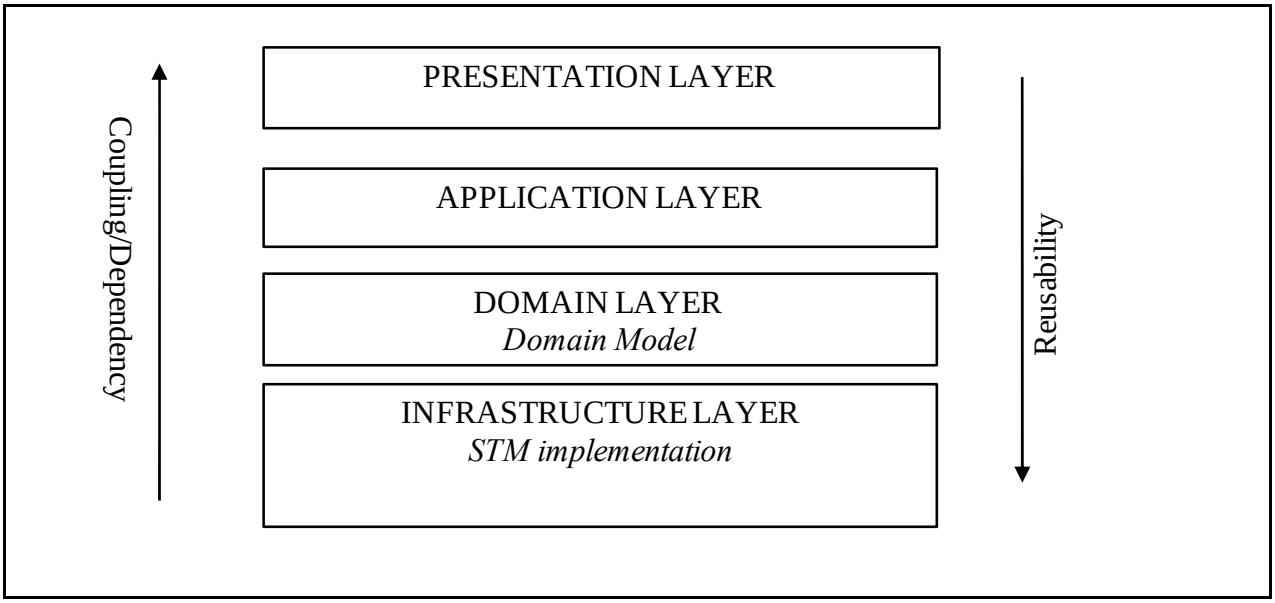

Figure 2 Layered Design pattern for QSTM.

Although the QSTM uses an STM, the design of the STM differs from the ones in [1] and [7]. The QSTM gets rid of the *Ownerships* vector and replaces it with a lock called the *commitLock* and uses *thread local quarantines* instead of a *log*. The UML class diagram in Figure 3 provides an overview of the layout for the QSTM design pattern. Furthermore, unlike the STM in [1], the QSTM does not need to know about the memory cells it is going to read/write in advance. The use of the thread local quarantines helps solve this issue making it *dynamic*.

4.2.1.1 STM

The STM is the *shared state store*. It behaves like the collection of *memory cells* whose contents can be modified by special operations called *transactions*. It is analogous to a thread-pool or manager.

The STM has a collection of *memory cells* called *memory.* The *shared state* is held in each of these memory cells. The STM acts as the *resource manager*, allowing the programmer to create and delete these memory cells. The STM also allows the programmer to submit the *transactional actions* to operate on the *shared state* held in its memory cells — update the contents of the memory cells. It has a *lock* called *commitLock*. This lock is used to synchronize the *transactions* when they are committing their updates to the STM's memory.

4.2.1.2 MemoryCell

The memory cell is the actual container that holds the *shared state.* The STM holds a collection of these memory cells — *memory*. The memory cells are concrete implementations of the *TVar* interface making them transactional variables. The *MemoryCell* is package-scoped in order to prevent accidental modification by non-STM operations. The *TVar* interface is used to give restricted access to the users of the QSTM implementation.

Each memory cell has a unique *ID* and *data*. The *data* is of *Value* type making it *equitable* and *cloneable.* This is needed because when we read the contents of a memory cell, we always return a copy — cloning level determined by the implementer — of the data. This prevents any accidental direct modification or corruption of the actual data.

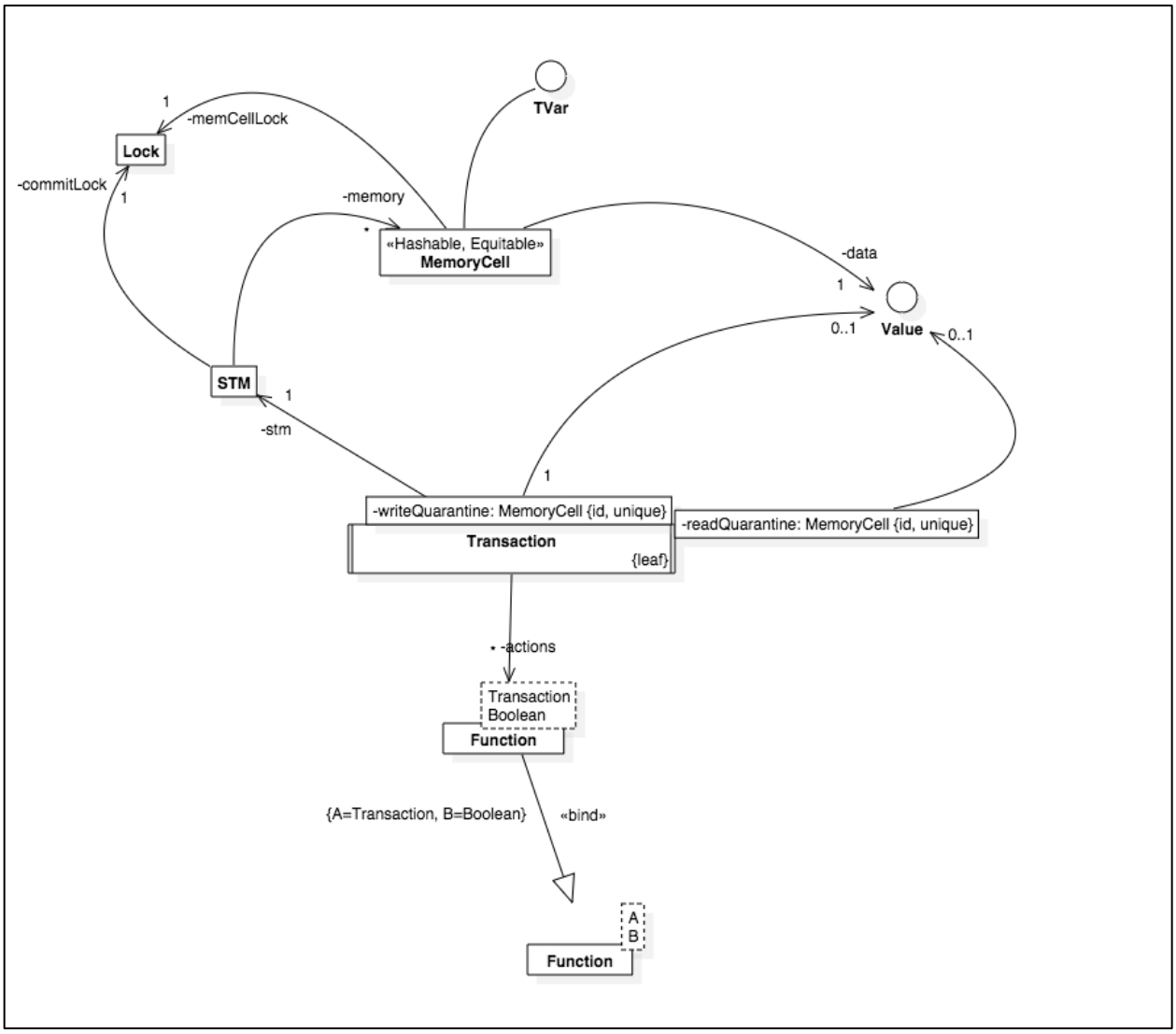

Figure 3 QSTM design pattern UML overview.

Moreover, the memory cell also has its own lock called *memCellLock.* This provides more granular synchronization. Although, this lock exists, it is visible only in the QSTM implementation layer and provides no hindrances while writing modular code using the QSTM implementation.

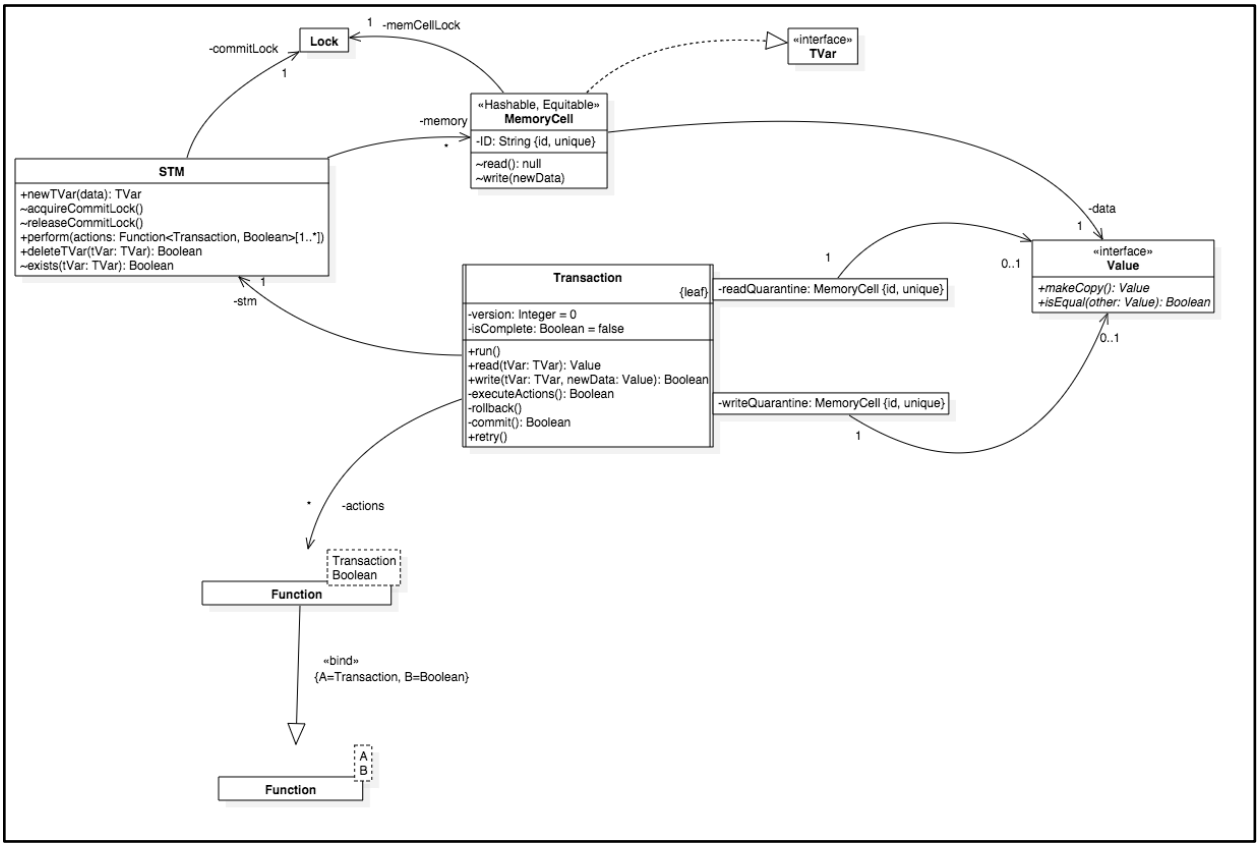

Figure 4 QSTM UML class diagram.

Moreover, the *MemoryCell* needs to be *hashable* and *equitable* since it is stored in the *thread local quarantines* (hash tables) of the transactions.

4.2.1.3 TVar

The *transactional variable* with contents. It is an *empty interface* that is visible to the consumers of the QSTM implementation. It ensures that the actual data in the memory cells is not polluted by the consumers. Also, it makes the resources being managed by the QSTM implementation explicit — the account state in Listing 5. The QSTM implementation should know how to convert the *TVar* into its internal concrete implementation for further processing.

4.2.1.4 Value

The *Value* interface represents any data that can be stored and managed by the QSTM implementation. Since the data being stored in the QSTM is actually contained in the *memory cells*, it is necessary for the data to be *cloneable* and *equitable.* For this reason, this interface enforces that the methods *makeCopy* and *isEqual* be implemented by the data intended to be stored and managed by the QSTM implementation.

Generally, the *State* part of the domain objects implement the *Value* interface as in Listing 4.

4.2.1.5 Transaction

The *Transaction* represents the main source concurrency in the QSTM. It is an *active object* that performs the *transactional actions* submitted to the STM. It is represented as a leaf level class that can no longer be extended. This is to ensure that no one overrides the behavior of a transaction's execution path by extending it — malicious extensions could lead to data corruption. Moreover, it is the only way to operate on the *shared state* being managed by the STM. The execution logic of the transaction is present in its *run()*.

QSTM's *transaction* unlike in [1, 7] does not maintain a *read/write set*. Instead, it maintains two *hash tables* called the *readQuarantine* and *writeQuarantine*. These hash tables map from a *memory cell* to the *value* contained in the memory cell at the time of read/write operation. This is also the reason for naming our STM pattern as *Quarantined STM* (QSTM). The use of these *thread local hash tables* (*quarantines*) ensures that the transaction executes in *isolation*. This also enables the transaction's results to be visible to all its peers at the same time

when it finally commits its changes to the STM. If a transaction fails to commit, its changes are not visible to its peers, making the execution of a transaction *atomic*.

The *transaction* maintains a version counter that is updated after every successful execution of the transaction. It also maintains a Boolean flag (*isComplete)* that indicates the execution status of the *transaction*. The *transaction* has a loop in its *run()* that keeps executing until its *isComplete* flag is set to *true* — which only happens upon a successful *commit phase*.

The execution logic of the transaction can be divided into two key phases:

1. *Execute Actions Phase*: In this phase, the transaction executes each of the *transactional actions* it was created with. All operations in this phase happen on the transaction's *quarantines*. If any of the transactional actions fails — returns *false*, the transaction fails and retries from the beginning. The failure of the transaction and its retrying from beginning is called *rollback*. When the transaction rolls back, it re-initializes its *quarantines* to clear all the results of computations from its last execution. This is done to ensure the most recent version of data is used for the current computation. After a successful *Execute Actions* phase, the transaction enters its *Commit* phase.
2. *Commit Phase*: In this phase, the transaction tries to update the actual contents of the memory cells it was operating upon. When a transaction enters its commit phase, it tries to acquire the *commitLock* on the STM. This ensures that no other transaction updates the STM's memory cells at the same time. It also ensures that the updates of the transaction are visible to its peers at once. When the transaction acquires the *commitLock*, it starts validating its *read-quarantined values*. During

this validation, it compares the values in its *readQuarantine* with the actual values of the memory cells. If the actual values have changed in the meantime, the transaction's commit phase fails, and it rolls back. If the validation phase of the transaction succeeds, it updates the contents of its *write-quarantined memory cells* with the values held in its *writeQuarantine*. Then, after updating the memory cells, the transaction releases the *commitLock* and the commit phase completes successfully. Upon successful completion of the commit phase, the transaction is marked as complete and its version counter is incremented.

All the heavy lifting of the transaction is done at the QSTM implementation layer and it does not leak into the domain layer. The STM class provides the necessary interface to perform the actions concurrently to the domain layer programmers.

When *transactional actions* are submitted to the STM, it creates a *transaction* with those actions and executes the *transaction*. A *transactional-action* is a *function* (Figure 4) that accepts a transaction instance and returns a *Boolean status*. The *status* is *true* if the action executed successfully, else it is *false*.

The motivation behind making a transactional action is inspired from [7, 10]. Since the transactions may roll back, it is necessary for them to only access data from their own *quarantines*. If they access any data not maintained by the STM or outside their quarantines, it will lead to data corruption when the transaction rolls back. In order to prevent this scenario, Haskell provides the STM *monad* [7] that prevents mixing of data not maintained by the STM with data maintained by the STM.

A *function* is one of the easiest ways of defining a local environment in most programming languages. By representing a *transactional action* as a *function* mapping from a transaction instance to a *Boolean status*, we can define the bounds of the data accesses. Although it is not strict enough like in case of *Haskell*, it provides enough structure to reduce careless mistakes. For example, the code in Listing 6 is a sample transaction action designed in *Java*.

The *transaction* exposes two public methods *read()* and *write()*. These methods enable the programmer to perform *transactional-reads* and *transactional-writes* on the memory cells.

- *read()*: The *read()* operation performs a *transactional-read*. Internally, it converts the transactional variable to get the concrete memory cell instance. Then, the *transaction* checks if the memory cell exists in its *read-quarantine*. If it exists, the value returned is the copy of the value in the *read-quarantine*. Otherwise, it reads the contents of the memory cell, stores the value in its *read-quarantine* and then returns back the copy of the value. The *read()*, always returns a copy of the value. This prevents any accidental modifications by any operations. Some implementations might prefer making a deep clone when cloning the values. However, cloning just the portion that might get affected is also acceptable. Once the memory cell has been populated into the *read-quarantine*, all subsequent reads happen from the *read-quarantine*.
- *write():* The *write()* operation performs a *transactional-write*. It writes to the transaction's *write-quarantine*. The contents of the transaction's *write-quarantine* are flushed to the actual memory cells of the STM upon successful validation in the *commit phase*.

```
/**
 * Creates a new transactional action that invokes the {@link TArray#add1001()}
 * on the instance stored in the STM.
 *
 * @param tvar
 *          The transactional variable that holds the TArray instance.
 *
 * @return The transactional action that invokes the {@link TArray#add1001()} on
 * the instance inside the transactional variable tvar.
 */
private static Function<Transaction, Boolean> performAdd1001(TVar tvar) {
    return t -> {
      TArray ta = t.read(tvar, TArray.class);
      ta.add1001();
      return t.write(tvar, ta);
    };
  }
```

Listing 7 A possible transactional action in Java.

The code sample in Listing 7 shows the creation of a *transactional action* and the usage of the *read()* and *write()* operations. A more detailed example will be introduced in Section 4.3 after discussing a possible *Java* implementation of the QSTM pattern.

4.2.2 Advantages of QSTM

- *High-level interface for concurrent programming*: The QSTM pattern language provides a higher-level interface for concurrent programming. It provides an abstraction over the layer that actually introduces concurrency and allows the programmer to have a sequential view of operations.
- *Move away from lock-based programming*: We have moved away from lock-based programming. The programmer no longer needs to worry about locks or other synchronization tools while developing the domain layer. The *transactional actions* provide a better interface and make the changes in state explicit.

- *Advanced language support*: If programming languages have the QSTM pattern language built into them, they can provide even better abstraction to the programmers.

4.2.3 Limitations of QSTM

- *Optimism causes always failing transactions*: QSTM pattern language uses an *optimistic* STM as the *state store*. Since, we have optimistic transactions, the transactions execute thinking they will not roll back. The decision for a roll back happens when the transactional action fails, or the commit phase validation fails. This leads to a special condition where there might be transactions that may never succeed, always failing and rolling back. A *pessimistic approach* might prevent this scenario but, it might reduce performance in return.
- *Increased memory consumption*: By adding an in-memory layer (or abstraction), we are increasing the memory consumption. Moreover, in languages like *Java* where *Threads* are heavy weight objects, having the *Transactions* of the QSTM be threads or *active objects* may use excessive memory. This however can always be solved by using a *thread-pool* [24] while implementing the QSTM's STM layer. In languages like *Go* [2] with lightweight threads — *goroutines* — having the transactions become *active objects* has no adverse impact.
- *Lack of IO*: Since the transactions in the QSTM implementation might fail and roll back, we cannot have the transactional actions affect any data/state not being managed by the STM (especially perform IO actions inside transactions). If they do, then the repeated roll backs might leave the outside data in an inconsistent state (or cause undesired IO behavior).

4.3 QSTM - JAVA IMPLEMENTATION

There are several possible implementations of the QSTM's STM layer. This section covers one such possible *Java* implementation of the QSTM pattern. The source code is hosted online at [25].

In this implementation, we have the *stm* package holding the classes and interfaces needed for the QSTM's STM layer implementation: *Value*, *TVar*, *MemoryCell*, *STM*, and *Transaction*. *Project Lombok's* [26] annotations are used for boilerplate reduction and code generation; *SLF4J* [27] and *Logback* [28] are used for logging. These are third party tools/libraries and can be replaced with other offering depending on the implementer's discretion.

4.3.1 Value

Value is implemented as an interface with two methods: *makeCopy* and *isEqual*. Any data that needs to be stored in the STM needs to implement this interface. The *makeCopy* method is used for making *working clones* of the data being stored in the STM — working clones are copies made from the object by copying only those attributes that can be modified when operating on the object. The *isEqual* method provides equitability, using it we can verify if two *values* are equal. This is of utmost importance when the transactions are validating the *read-quarantined values*. The *Java* definition is provided in Listing 8.

4.3.2 TVar

TVar is implemented as an *empty interface*. This is done deliberately to prevent the consumer of the QSTM's STM implementation to directly access and modify the contents of the

memory cells. The implementation knows how to convert the *TVar* into a concrete memory cell and does so internally. The *Java* class definition is provided in Listing 9.

```
/**
 * A value that can be stored in the STM's memory cell. It is cloneable and
 * equitable.
 *
 * @author sidmishraw
 *      Qualified Name: stm.Value
 */
public interface Value {

  /**
   * Creates and returns a clone/copy of the object whose modification doesn't
   * affect the original object.
   *
   * @return The zero modification impact clone of the object.
   */
  Value makeCopy();

  /**
   * Checks if this value is equal to the given value.
   *
   * @param v
   *          The value to equate against.
   *
   * @return true if both are equal, else false.
   */
  Boolean isEqual(Value v);
}
```

Listing 8 Value interface definition in Java.

```
/**
 * The transactional variable with contents of type Value.
 *
 * @author sidmishraw
 *         Qualified Name: stm.TVar
 */
public interface TVar {}
```

Listing 9 TVar interface definition in Java.

4.3.3 MemoryCell

MemoryCell is a package-scoped *class* that implements the *TVar* interface. It represents the actual memory cell being managed by the *STM*. The memory cell has a unique ID of type *UUID* (universally unique identifier). *Java's UUID* utility class [29] is used to generate random *type 4 UUIDs* when constructing a new memory cell.

The data contained in the memory cell is of type *Value*. The memory cell also has a *ReentrantReadWriteLock* [30] called *memCellLock*. This read-write lock provides even more granular control over the contents of the memory cell when reading from or writing to it. The memCellLock is not exposed and is used internally by the memory cell.

The data of the memory cell can be read by invoking its *read()*. When invoked, the *read()* of the memory cell acquires the *memCellLock* in *READ_MODE* and then returns the copy of the actual data before releasing the *memCellLock*.

```
/**
 * Reads the data in the memory cell.
 * This method is package scoped for security reasons.
 *
 * @return The copy of the data contained in the memory cell.
 */
Value read() {
  Value data = null;
  try {
    this.memCellLock.readLock().lock();
    data = this.data.makeCopy();
    return data;
  } finally {
    this.memCellLock.readLock().unlock();
  }
}
```

Listing 10 MemoryCell's read() definition.

The data of the memory cell can be updated by invoking its *write()*. When invoked, the memory cell acquires the *memCellLock* in *WRITE_MODE* and then overwrites the existing data with the new data before releasing the *memCellLock*.

```
/**
 * Writes the data into the memory cell.
 * This method is package scoped for security reasons.
 *
 * @param newData The new data to be written into the memory cell.
 */
void write(Value newData) {
  if (Objects.isNull(newData)) {
    return;
  }
  try {
    this.memCellLock.writeLock().lock();
    this.data = newData;
  } finally {
    this.memCellLock.writeLock().unlock();
  }
}
```

Listing 11 MemoryCell's write() definition.

The STM implementation layer handles conversion from *TVar* to *MemoryCell* when required.

4.3.4 Transaction

Transaction is as a *Java Runnable* [31]. It is a *final* class in order to prevent the consumer of the QSTM implementation layer from adding faulty logic — security against external modifications.

It has package-scoped attributes named *version* and *isComplete*. The *version* of the *transaction* is incremented upon a successful *commit phase*, and the *isComplete* flag is used to indicate if the *transaction* is complete. With the QSTM allowing for deletion of memory cells,

the transaction also has a flag named *shouldAbort*. When *shouldAbort* is set, the transaction is invalidated and aborts without retrying.

The *readQuarantine* and *writeQuarantine* of the transaction are *hash tables* implemented using *Java's HashMap* [32]. The *MemoryCell* becomes the *key* and *Value* becomes the *value* of these *hash tables*. Moreover, *Project Lombok's* [26] *@Builder annotation* creates a builder API for constructing transactions making the API cleaner and elegant.

```
/**
 * Creates a new transaction for the given STM.
 *
 * @param stm
 *          the STM object that the transaction operates on.
 */
@Builder
Transaction(STM stm, @Singular List<Function<Transaction, Boolean>> actions) {
  this.version = 0;
  this.isComplete = false;
  this.readQuarantine = new HashMap<>();
  this.writeQuarantine = new HashMap<>();
  this.stm = stm;
  this.actions = actions;
  this.shouldAbort = false;
}
```

Listing 12 Transaction construction and @Builder annotation from Project Lombok.

The *run()* has a loop with the entry condition being true only when the *isComplete* and *shouldAbort* flags are not set. This loop makes the transaction keep retrying until it either succeeds or is invalidated.

When the loop begins its execution, the transaction first executes all the *transactional actions*. It does so by invoking the *executeActions()*. If it fails, the transaction rolls back by invoking its *rollback()*. Otherwise, it enters its *commit phase* and invokes the *commit()*. If the *commit phase* is successful, it sets its *isComplete* flag to *true* and breaks out of the loop; it

increments its version and completes its execution. Otherwise, it rolls back and retries from the beginning.

Although the transaction's *run()* should not be visible to the consumer, having implemented *Java's Runnable* interface, the run() remains public scoped. However, the *Go* implementation keeps the *run()* package-scoped, hiding it from the consumer.

Apart from *run()*, the transaction provides two public methods: *read* and *write*. These methods are used when creating *transactional actions*. To demonstrate their usage, we'll expand the definitions of the *deposit()* in Listing 13, *withdraw()* in Listing 14, and *transfer()* in Listing 15 of the *Account class* introduced in Listing 5.

```
/**
 * Deposits the amount into this account.
 *
 * @param amount
 *          The amount to be deposited into the account.
 */
@SuppressWarnings("unchecked")
public void deposit(Integer amount) {

  this.stm.perform((Transaction t) -> {

    AccountState as = t.read(this.accountState, AccountState.class);

    as.deposit(amount);

    return t.write(this.accountState, as);
  });

}
```

Listing 13 Account's deposit() implemented using Java QSTM implementation.

As shown in Listing 13, we can define a *transactional-action* as a *Java 8 Lambda* with transaction as input and *Boolean* status as output. We use the *read()* of the *transaction* to read the contents of a *transactional-variable*. After operating on the contents of the

transactional-variable, we can write it back into the *transactional-variable* using the transaction's *write()*. Notice the programming style, it is still sequential, and we do not have to worry about synchronization; the synchronization is being handled by the QSTM implementation layer. We can handle any erroneous situations inside the *transactional-actions* and cause it to fail and retry by returning *false* from the *transactional-actions*. In languages like *Ruby* [33] where we can pass around blocks of code, we could have an operation on the transaction object called *retry()* which, when invoked, could break the execution of the *transactional-action* and cause a retry. The source code in Listing 14 showcases the usage of the *false-returning-retry -mechanism*.

```
/**
 * Withdraws the specified amount from this account.
 *
 * @param amount
 *          The amount to be withdrawn from the account.
 */
@SuppressWarnings("unchecked")
public void withdraw(Integer amount) {

  this.stm.perform((Transaction t) -> {

    AccountState as = t.read(this.accountState, AccountState.class);

    try {

      as.withdraw(amount);

      return t.write(this.accountState, as);

    } catch (Exception e) {

      return false;

    }
  });

}
```

Listing 14 Returning false from a transactional action in Account's withdraw() causes it to retry.

Finally, the *transaction* ensures that the *transactional-actions* managed by it are executed *atomically*. So, if any one of them fails, the entire execution fails and the transaction rolls back. This makes an ideal use-case for atomic actions such as bank account transfers. The source code in Listing 15 showcases the *transfer()* definition of the *Account* class in Listing 5.

```
/**
 * Transfers the desired amount from this account to the destination account.
 *
 * @param destination
 *          The account to transfer to.
 * @param amt
 *          The amount to transfer.
 */
@SuppressWarnings("unchecked")
public void transfer(Account destination, Integer amt) {

  this.stm.perform((Transaction t) -> {

    AccountState srcState = t.read(this.accountState, AccountState.class);
    AccountState destState = t.read(destination.accountState, AccountState.class);

    try {

      srcState.withdraw(amt);
      destState.deposit(amt);

      Boolean status = t.write(this.accountState, srcState);
      status = status && t.write(destination.accountState, destState);

      return status;

    } catch (Exception e) {

      return false;

    }

  });

}
```

Listing 15 Account's transfer() is an ideal usage for the atomic transactional actions.

Another reason for hiding the transaction construction from the consumer is *simplicity*. By having the consumer only interact with one class to tackle all their use-cases, it reduces the

cognitive load. In this implementation, the consumer only needs to interface with the *STM* instance to achieve their tasks. By abstracting the construction of the transaction object, we are able to provide them with a cleaner and simpler API.

4.3.5 STM

The *STM* in this implementation is a public scoped class with a private list of *MemoryCells* called *memory*. *Java's List* and *ArrayList* from its *Collections framework* [34] are used to implement the *memory*. The *commitLock* of the STM is a lock implemented as *Java's ReenterantLock*.

The STM is *fat* and is the only interface needed by the consumer of the QSTM implementation to interact with it. It exposes three public scoped methods for the consumer to use.

1. *newTVar(Value)*: This method is used for creating new *transactional-variables*. When invoked, it first creates a *MemoryCell* and initializes it with the desired data. Then, it adds the memory cell to the STM's memory and then returns the *TVar* reference to it.
2. *deleteTVar(TVar)*: This method deletes the *transactional-variable* from the STM's *memory* by removing it from the list. Once the memory cell has been removed from the memory, all the transactions dependent on it are invalidated and aborted. Although the memory cell is removed from the STM and can no longer be used in *transactional-actions*, it only gets garbage collected at the JVM's discretion. Once a *TVar* is deleted, we can reuse the reference to point to some other *transactional-variable*.
3. *perform(Function<Transaction, Boolean>)*: This method is used to submit *transactional-actions* to the STM to perform. In this implementation, we spawn a new

transaction for each call to the *perform()*. The *transactional-actions* submitted together are added to the same *transaction* and are executed sequentially on the same thread. So, to have *transactional-actions* execute concurrently, they should be submitted to the STM separately — using separate invocations of the method. Another alternative could be to maintain a fixed count *thread-pool*. Then, we could submit the transactions to the thread-pool and reuse the threads. Since the transaction implements the *Runnable* interface, it can be easily implemented using the thread-pool (*Java's ExecutorService*) [24].

```
/**
 * The STM spins up a transaction to perform the actions.
 *
 * @param actions
 *          The actions to perform transactionally.
 */
@SuppressWarnings("unchecked")
public void perform(Function<Transaction, Boolean>... actions) {

  List<Function<Transaction, Boolean>> transactionalActions
      = Arrays.asList(actions);

  // Build a new transaction for executing the
  // transactional actions submitted.
  //
  Transaction t = Transaction.builder()
      .stm(this)
      .actions(transactionalActions)
      .build();

  // Execute the transaction
  //
  t.execute();

}
```

Listing 16 perform() for submitting transactional actions to the STM.

The code in Listing 17 shows the driver code for using the *Account* class implemented using the QSTM pattern; the output logs are shown in Listing 18.

```
public static void main(String[] args) {

    // Create two accounts `account1` and `account2` with initial balances
    // `100` and `200` respectively.
    //
    Account acc1 = Account.builder().accountName("Account1").initialBalance(100).stm(stm).build();
    Account acc2 = Account.builder().accountName("Account2").initialBalance(200).stm(stm).build();

    // print the JSON structure for acc1 and acc2, cheap debugging.
    //
    logger.info("Account 1:\n" + acc1.toString());
    logger.info("Account 2:\n" + acc2.toString());
    stm.printState();

    // Create a threadPool having 4 threads to simulate multiple threads
    // requesting changes on the accounts.
    //
    ExecutorService threadPool = Executors.newFixedThreadPool(4);

    // perform the actions on the accounts
    // deposit 50 into acc1: acc1 = 150
    // deposit 300 into acc2: acc2 = 500
    // transfer 35 from acc1 to acc2: acc1 = 150 - 35 = 115, acc2 = 500 + 35 = 535
    // transfer 50 from acc2 to acc1: acc1 = 115 + 50 = 165, acc2 = 535 - 50 = 485
    //
    // Finally, acc1 = 165, acc2 = 485 (Test for consistency of the operations)
    //
    // Note: all these actions happen to be running on separate threads under the hood.
    //
    threadPool.submit(() -> acc1.deposit(50));
    threadPool.submit(() -> acc2.deposit(300));
    threadPool.submit(() -> acc1.transfer(acc2, 35));
    threadPool.submit(() -> acc2.transfer(acc1, 50));

    // shutdown the threadPool, its job is done
    //
    shutdown(threadPool);

    // Hold for input, an easy way to make the main thread wait.
    // Other implementations might include use of CountDownLatch or Thread#join().
    //
    try (Scanner sc = new Scanner(System.in)) {
      sc.nextLine();
    } catch (Exception e) {}

    // Poor man's debugging -- printing the JSON structures to check the current state
    // of the accounts.
    //
    logger.info("Account 1:\n" + acc1.toString());
    logger.info("Account 2:\n" + acc2.toString());
    stm.printState();

}
```

Listing 17 Driver code for using Account class defined using QSTM pattern.

```
  -------- ABRIDGED LOG ------- START ---------------

  11:05:09.628 [main] DEBUG stm.STM - HARMLESS :: DEBUGGING STATE :: {
    "memory": [
      {
        "ID": "66732a4b-3f16-4d05-8ae8-c85a1f43d547",
        "data": {
          "balance": 100
        }
      },
      {
        "ID": "ef9c5e2a-a10f-4de6-9bad-1899d8080f0d",
        "data": {
          "balance": 200
        }
      }
    ]
  }
  11:05:11.434 [Thread-1] DEBUG stm.Transaction - Transaction: Thread-1 has started execution.
  11:05:11.434 [Thread-0] DEBUG stm.Transaction - Transaction: Thread-0 has started execution.
  11:05:11.437 [Thread-2] DEBUG stm.Transaction - Transaction: Thread-2 has started execution.
  11:05:11.437 [Thread-3] DEBUG stm.Transaction - Transaction: Thread-3 has started execution.
  11:05:12.760 [Thread-3] INFO stm.Transaction - Thread-3 begins its commit phase
  11:05:12.760 [Thread-1] INFO stm.Transaction - Thread-1 begins its commit phase
  11:05:12.760 [Thread-2] INFO stm.Transaction - Thread-2 begins its commit phase
  11:05:12.760 [Thread-3] INFO stm.Transaction - Thread-3 begins validating its read quarantined values in the
commit phase
  11:05:12.760 [Thread-0] INFO stm.Transaction - Thread-0 begins its commit phase
  11:05:12.761 [Thread-3] INFO stm.Transaction - Thread-3 ends its commit phase
  11:05:12.761 [Thread-3] DEBUG stm.Transaction - Transaction: Thread-3 has finished execution.

  --------------------------- CONTENT NOT SHOWN FOR SAKE OF BREVITY --------------------

  11:05:12.762 [Thread-2] INFO stm.Transaction - Thread-2 ends its commit phase

  11:05:12.762 [Thread-2] DEBUG stm.Transaction - Transaction: Thread-2 has finished execution.

  --------------------------- CONTENT NOT SHOWN FOR SAKE OF BREVITY --------------------

  11:05:16.199 [main] DEBUG stm.STM - HARMLESS :: DEBUGGING STATE :: {
    "memory": [
      {
        "ID": "66732a4b-3f16-4d05-8ae8-c85a1f43d547",
        "data": {
          "balance": 165
        }
      },
      {
        "ID": "ef9c5e2a-a10f-4de6-9bad-1899d8080f0d",
        "data": {
          "balance": 485
        }
      }
    ]
  }

  -------- ABRIDGED LOG ------- END ---------------
```

Listing 18 Logs for the execution of driver code in Listing 15.

The logs in Listing 18 are abridged but, they show that each transaction is launched as a separate thread and all of these threads are accessing the same *shared state store* — the STM. The *Go* implementation of the QSTM pattern is covered in Section 4.4.

4.4 QSTM – GO IMPLEMENTATION

This section covers a possible *Go* implementation of the QSTM pattern. The source code is hosted online at [35]. Since the implementation follows the QSTM pattern language, the names of the components remain the same. This section will discuss the language specific differences between *Go* and *Java* and how the *Go* implementation, although different from the *Java* implementation still conforms to the QSTM pattern.

Unlike *Java*, *Go* is not object-oriented, so, it does not have *classes*. However, classes can be emulated using *Go's structures*. The QSTM's STM implementation layer is located inside the *stm* package. The classes (*STM*, *Transaction*, and *MemoryCell*) are implemented as *Go structs*. *Value* and *TVar* are implemented as *Go interfaces*.

TVar is implemented as an empty interface similar to the *Java* implementation in Section 4.3. Similarly, *Value* is an interface having two methods (*MakeCopy* and *IsEqual*) with definitions similar to the *Java* implementation. The source code snippet in Listing 19 shows the definitions of *TVar* and *Value* interfaces in *Go*. The naming convention is *Go* is different from *Java* and the casing of the identifier names affects their visibility — lower-cased identifiers are package-scoped and title-cased identifiers are public scoped.

```
package stm

// TVar is the transactional variable. It is an empty interface that is used as
// the reference to memory cells by the end consumer. This is to prevent the consumer
// from directly modifying the contents of the memory cells.
type TVar interface{}

// Value is any value that can be stored in a memory cell.
// Value can be stored in the STM.
type Value interface {
    MakeCopy() Value             // MakeCopy makes a deep copy of the Value.
    IsEqual(other Value) bool    // IsEqual checks for the equality between two values.
}
```

Listing 19 Go definitions for TVar and Value interfaces.

The *MemoryCell* is implemented as a package-scoped struct. It is made package-scoped using lower-casing for its name. The code snippet in Listing 20 shows the definition of the *MemoryCell* class in *Go*.

```
package stm

// memoryCell represents a memory cell where the data is stored.
type memoryCell struct {
    id          string       // The unique identity of the memory cell.
    data        Value        // The contents of the memory cell.
    memCellLock *sync.RWMutex // A read-write lock for obtaining more granular locking.
}
```

Listing 20 MemoryCell defined in Go.

Unlike in *Java*, we store pointers to memory cells in the STM. The *memCellLock* is a pointer to a read-write mutex — *RWMutex* — from *Go's sync* package. Since, we are trying to

hide the memory cell implementation from the consumer, all the methods on this structure are package-scoped as well — have lower-case identifiers.

```
package stm

// STM is the single shared memory store that can only be modified by transactions.
type STM struct {
    memory     []*memoryCell // the collection of memory cells makes up the memory
    commitLock *sync.Mutex   // the global commit lock
}

// New makes and initializes a new STM instance.
func New() (stm *STM) {
    stm = new(STM)
    stm.memory = make([]*memoryCell, 0, 0)
    stm.commitLock = new(sync.Mutex)
    return stm
}

// NewTVar creates a new memory cell in the STM and returns the reference
// to the memory cell as a TVar instance.
func (stm *STM) NewTVar(data Value) TVar {
    memCell := newMemCell(data)
    stm.memory = append(stm.memory, memCell)
    return TVar(memCell)
}

// Perform accepts the transactional actions submitted to the STM and performs them.
func (stm *STM) Perform(actions ...func(*Transaction) bool) {
    for _, action := range actions {
        t := newTransaction(stm, action)
        t.execute()
    }
}
```

Listing 21 STM definition in Go.

The *STM* is a *Go struct*. It has a *slice* of pointers to memory cells called *memory* and a *commitLock* which is a *mutual exclusion lock* (*sync.Mutex*). It exposes two public operations to create *transactional-variables* and perform *transactional-actions*. This implementation does not

allow deletion of memory cells. It can be viewed as a scenario specific implementation where the consumers never desire to delete state data. Also, this implementation creates a new transaction for each *transactional-action* submitted to the STM. This is a variant of the STM's *perform* method in the *Java* implementation in Section 4.3. This ensures that all the *transactional-actions* are run on separate threads concurrently.

In *Go*, concurrency is achieved by using lightweight threads called *goroutines* [36]. The *Transaction* is implemented as a public scoped *struct*. It has an integer *version*, *bool* flag called *isComplete*, a function mapping from *Transaction* to *bool* called *action*. It also has two *quarantines* that are hash-tables implemented as *Go map* mapping from a pointer to *memoryCell* to a *Value*. Since the pointers are basically memory addresses, they can be readily hashed by the *Go* runtime and we do not need to implement any functions to generate hashes explicitly. Finally, it has a pointer to the *STM* instance it is going to operate upon. Listing 23 shows the definition of the *Transaction struct* in *Go*.

```
package stm

// Execute executes this transaction as another thread.
// Package scoped, not visible outside the stm implementation layer.
//
func (t *Transaction) execute() {
    go t.run()
}
```

Listing 22 Executing Transaction on a goroutine.

A transaction is executed on a separate thread of control — *goroutine* — by invoking its *execute()* method. When the *execute()* is invoked, the transaction's *run()* method is executed on a

goroutine, as shown in Listing 22. Unlike *Java* implementation's public scoped *run()*, this implementation has a package-scoped *run()* enabling better encapsulation and abstraction.

```
package stm

// Transaction is the only way to modify the memory cells in the STM.
type Transaction struct {
    version         int                     // version of the transaction
    isComplete      bool                    // flag showing if the transaction is running
                                            // or is complete
    action          func(*Transaction) bool // the action that this transaction executes
    readQuarantine  map[*memoryCell]Value   // the read quarantine
    writeQuarantine map[*memoryCell]Value   // the write quarantine
    stm             *STM                    // the reference to the STM this transaction
                                            // intends to modify
}

// Reads the contents of the memory cell referenced by the `tVar`.
func (t *Transaction) Read(tVar TVar) Value { . . . }

// Writes the new Data into the write quarantine.
// This will be flushed into the STM upon successful commit.
func (t *Transaction) Write(tVar TVar, newData Value) bool { . . . }

// execute executes this transaction as another thread – on a goroutine.
func (t *Transaction) execute() { . . . }

// The actual execution logic of the transaction.
func (t *Transaction) run() { . . .}

// rollback rolls back the transaction to the initial state so that it can retry.
func (t *Transaction) rollback() { . . . }

// commit flushes the contents of the write quarantine memory cells into the STM.
func (t *Transaction) commit() bool { . . . }
```

Listing 23 Transaction definition in Go.

The execution logic of the transaction remains unchanged from the *Java* implementation albeit the syntactic changes between *Java* and *Go*.

```
package main

import (
    "fmt"
    "github.com/sidmishraw/gostm/account"
    "github.com/sidmishraw/gostm/stm"
)

func main() {
    STM := stm.New()
    acc1 := account.NewAccount("account1", 100, STM)
    acc2 := account.NewAccount("account1", 500, STM)

    // Initial state of the STM [100, 500].
    STM.PrintState()

    // 1st transaction that transfers 100 from account2 to account1
    acc2.Transfer(acc1, 100)
    // 2nd transaction that transfers 10 from account1 to account2
    acc1.Transfer(acc2, 10)

    var k int
    fmt.Scan(&k)

    // final consistent state is going to be [190, 410].
    STM.PrintState()
}

// Output: cmd >> go run main.go
//
2018/04/19 12:50:03 ------- State of the STM --------
2018/04/19 12:50:03 {"id": 5cc2d196-0ff9-4f0b-9976-9577987e5ee5, "data": &{100}}
2018/04/19 12:50:03 {"id": 48d30abd-43bf-43d8-904f-17737a538894, "data": &{500}}
2018/04/19 12:50:03 ------- END --------
1
2018/04/19 12:50:06 ------- State of the STM --------
2018/04/19 12:50:06 {"id": 5cc2d196-0ff9-4f0b-9976-9577987e5ee5, "data": &{190}}
2018/04/19 12:50:06 {"id": 48d30abd-43bf-43d8-904f-17737a538894, "data": &{410}}
2018/04/19 12:50:06 ------- END --------
```

Listing 24 Example driver for QSTM implementation in Go.

Go's goroutine is lighter than *Java's Thread* and alleviates the need of a thread-pool but an alternate implementation using the thread-pool model to achieve concurrency could also be

proposed. Listing 24 shows the driver code and output logs for the *Go* implementation. The example used is a port of the same *Account* object introduced in Listing 5.

The intention behind the implementations in *Java* and *Go* is to showcase the fact that the QSTM pattern language is versatile and can be implemented in languages from different paradigms.

5 MONADS

Monads have been used to purify impure functions in pure functional programming languages. In *Haskell* [10], *monad* is implemented as the *Monad* type-class but, looking closely, a *monad* can be thought of as a generic pattern that is not limited to functional programming languages. Section 5.1 provides a brief background about monads and their usage. The languages of choice will be *Haskell* and *Scala* [17]. Then, Section 5.2 introduces the idea of *Monad* as a pattern and provides a pattern language for it. Finally, Section 5.3 introduces the *monad pattern* into the QSTM pattern covered in Section 4.2 and shows how monads fit easily into the QSTM pattern.

5.1 BACKGROUND

A *pure function* is defined as a function whose output depends only on its input. In pure functional programming languages like *Haskell*, all functions must be pure. However, in the real world, having just pure functions does not make sense. Pure functions cannot modify state of anything outside their execution context, so operations like input-output (IO), network communication, etc. become impossible. Any function that performs an action like IO, change in external state, etc. is actually performing a side-effect apart from producing the output. We use monads to represent these implicit side-effects. Once these side-effects become explicitly known, the impure function gets purified.

The *Java* source in Listing 25 is a definition of the pure function/static method *successor* that provides the successor of the given integer. The *impureSuccessor* in Listing 26 provides the successor, but it also produces a side-effect — prints to the *stdout.* Although it is common to see

method/function definitions similar to *impureSuccessor* in imperative languages, it is difficult to implement such style in pure functional languages like *Haskell*.

```
/**
 * Returns the successor of the desired integer.
 * It is a pure function since its output solely depends upon its input.
 *
 * @param x The desired integer.
 *
 * @return The successor of the integer provided.
 */
 public static Integer successor(Integer x) {
   return x + 1;
 }
```

Listing 25 A pure function.

```
/**
 * Returns the successor of the desired integer but, first
 * prints the successor to the standard output (stdout).
 *
 * @param x The desired integer.
 *
 * @return The successor of the integer provided.
 */
public static Integer impureSuccessor(Integer x) {
  System.out.println("successor = " + (x + 1));
  return x + 1;
}
```

Listing 26 An impure function/method.

So, to purify the side-effect introduced by functions like *impureSuccessor*, Haskell introduced the notion of a monad. Wadler in [37] introduces a type for computations/actions where the actions could be modifying an external state, IO, network communication, raising exceptions, etc. Basically, the implicit side-effect is given a type and that type is the *monad*. The source in Listing 27 introduces a *monad* to make the implicit IO side-effect of *impureSuccessor*

explicit. The definition of IO *monad* will be covered in Section 5.2 after the introduction of *monad* as a pattern.

```
/**
 * Purified version of the {@link MonadSimulator#impureSuccessor()}.
 * This method/function takes in the desired integer but,
 * returns an IO action. This action when performed will
 * cause the desired side-effect and return the successor of the
 * integer.
 *
 * @param x The desired integer.
 *
 * @return An IO action which when performed prints to stdout and
 * then returns the successor.
 */
public static IO<Integer> monadicSuccessor(Integer x) {
  return new IO<>(() -> {
    System.out.println("successor = " + (x + 1));
    return x + 1;
  });
}
```

Listing 27 Monads make implicit side-effects explicit.

5.1.1 Formal Definition

Formally, a *monad* is defined as a *triple* (*M*, *unit*, *) [37]. The first member of the triple is the *type constructor M*. It is followed by the operations *unit* and *bind(*)*. Wadler in [37] defines the type *M* or monad to be the type for computations or actions. The operation *unit* is defined as a function that takes a value and returns an action which, when performed, returns that value. The operation *bind(*)* is defined as a function that takes in a *monad* [*M a*] and applies a *transformer* function of the signature [$f :: a \rightarrow M\ b$] on it to produce another *monad* [*M b*]. The *Haskell* snippet in Listing 28 shows the function signatures for *unit* and *bind* operations of a *monad*.

```
-- The unit operation that takes a value of type `a`
-- and then returns an action of type `M` which when performed
-- returns that value.
unit :: a -> M a


-- The bind (*) operation that applies a transformer on a monad
bind :: M a -> (a -> M b) -> M b
```

Listing 28 The unit and bind operations.

5.1.2 Function composition

Function composition (.) is one of the key patterns used in functional programming to achieve modularity and code-reusability. It is a way to combine two pure functions to create a new pure function which, when applied to the input, produces output as if the individual functions were applied in-order. The UML activity diagram in Figure 5 shows function composition of two functions *f* and *g* to produce a new function *h*.

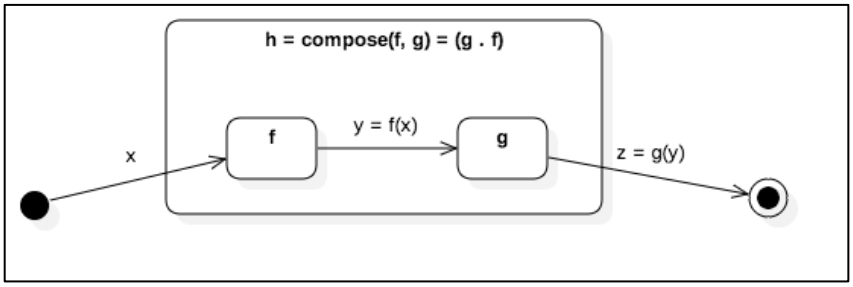

Figure 5 UML activity diagram for function composition.

Although *data-flow diagrams* are often used for functional programming models, the UML *activity diagrams* like in Figure 5 can be used when we want more generic models. Certain functional languages like *Haskell* also have special syntactic features to support function composition – *points free style* [38].

5.1.3 Bind

For *function composition*, we often assume that the functions are pure, they take one input, and they produce one output. However, when we have two impure functions, *function composition* is not easy. *Bind (*)* is the pattern used in this scenario. The *bind* pattern combines two impure functions to produce a new impure function which, when applied to the input, will produce the same effects as if the individual functions were applied in order [39]. The *activity diagram* in Figure 6 shows the *bind pattern* where two impure functions *f* and *g* are combined to form the new impure function *h*.

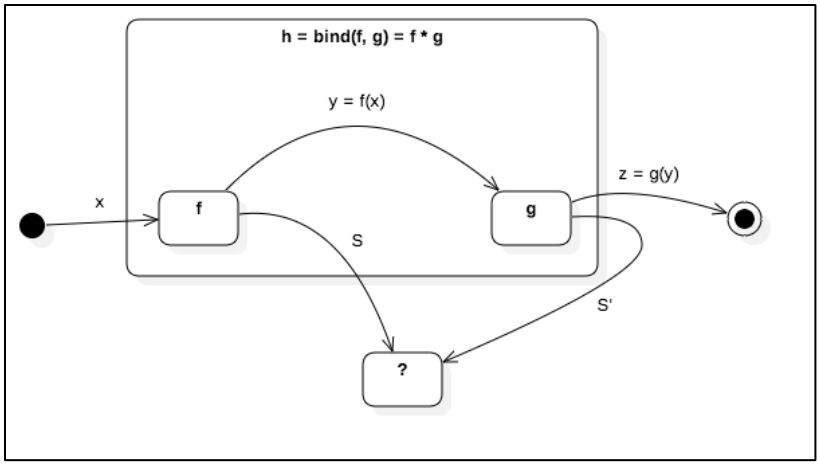

Figure 6 UML activity diagram for bind pattern.

Unlike in the *function composition* case in Section 5.1.2, the impure functions *f* and *g* have side-effects *S* and *S'* on something external to their execution contexts. The external resource could be IO streams, shared state, etc. In addition to the implicit side-effects, the functions *f* and *g* also produce explicit outputs. Moreover, the side-effects of the functions being bound together might be different, so it depends on the programmer to wire the implicit and

explicit outputs of these functions – reduces the code reusability. Designing language features for generic bind is difficult unlike function composition.

5.1.4 Monadic bind

The *bind pattern* can be extended to use *monads* for representing the implicit side-effects of the impure functions. In this pattern Figure 7, we purify the impure functions *f* and *g* by representing their outputs as *monads M*. The *monad* becomes a bundle of the implicit side-effect/action and the explicit result or data. It could be seen as a container whose contents can only be obtained after performing some side-effects to the external resource.

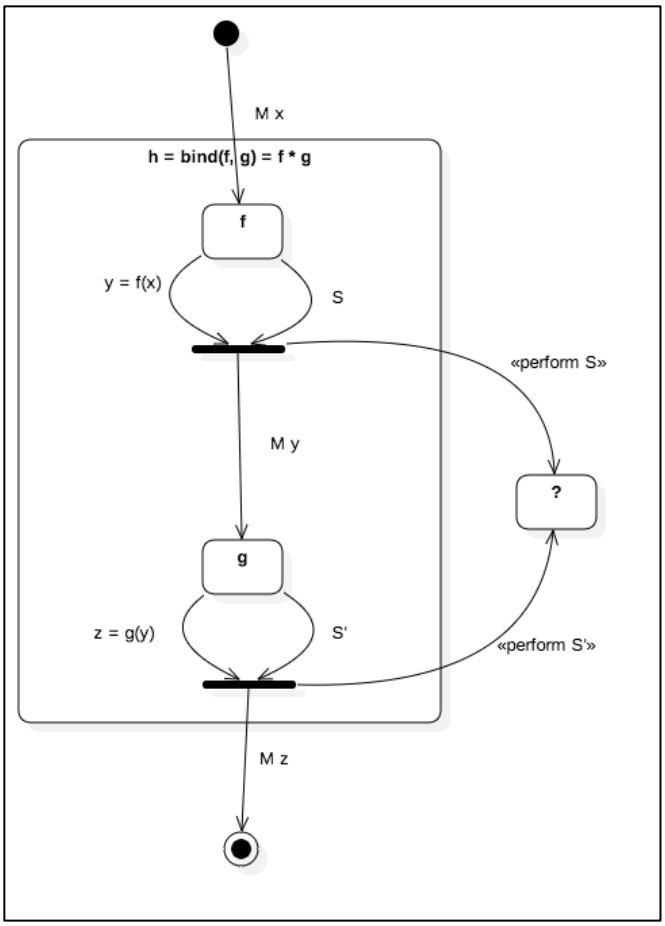

Figure 7 UML activity diagram for monadic bind pattern.

The introduction of a *monad* makes the computation *one-step-lazy*. The implicit side effect caused by the action *S* and *S'* are made explicit by preventing them from executing immediately. Since the execution becomes *one-step-lazy*, we get a *thunk* or *action object* that when performed or *unwrapped* will cause the side-effect and give us the explicit result. Also, with the introduction of *monads*, the functions *f* and *g* have been purified and their signatures look similar to the ones in Section 5.1.2. So, the *monadic bind* can be viewed as *function composition* for impure functions.

It is difficult to express *monads* without a well-developed data type system in the language. For instance, the type system of *Java* is not mature enough to define a generic monad like *Haskell*. With the lack of *Algebraic Data Types* (ADTs) it becomes difficult to express these pure functional concepts in OO languages like *Java* – although it can be achieved using *Scala*.

However, there are several ways to realize this pattern – although not through pure OO. One way could be to model an action/side-effect type as an object. When the action is performed, it produces the desired side-effect and provides us with the explicit value as promised – in *Java* *IO<String>* would represent an IO action that when performed would interact with the IO streams and return a *String* result.

5.2 MONAD PATTERN

The *monad* can be extended into a design pattern to target use-cases where we need to make side-effects/actions explicit. The pattern language is listed in Listing 29. The *builder pattern* [40] is an OO design pattern that closely resembles monad pattern. Like the builder pattern, the *monad pattern* is a *creational pattern* that lets the programmer create explicit actions and compose them together to build larger actions – *thunks*.

```
Name: Monad
Context: You are dealing with functions that have implicit side-effects and you want to
compose them together. The functions could be operations on objects or standalone functions.
Problem: Composing impure functions is not as straightforward as composing pure functions.
Also, writing a generic bind operation is difficult since the implicit side-effects of the
impure functions require special attention.
Solution: A monad is a triple (M, unit, *). First, represent the implicit side-effect as an
object or new type. Second, define the unit operation for this new type. The unit operation
should be able to convert any standalone value into a computation that can be performed to
give back the standalone value. Third, provide an operation to perform the side-effect
explicitly or unwrap the contents being held in the monad. Finally, define the bind operation
for the new type which can apply a transformer function to it and transform its side-effects
or contained value.
Use-cases: When there is a need to make implicit side-effects/actions explicit. Also, the
Monad pattern is best suited for scenarios where actions need to be chained/composed
together.
Consider these patterns next: Builder pattern, Function composition, Function bind
```

Listing 29 Monad pattern language.

The monad pattern makes the computation with side-effect *one-step-lazy* by wrapping it in another computation. To illustrate this pattern, we will define the IO *monad* used in the *monadicSuccessor* function given in Listing 27. We will be using *Java* to implement the *monad*.

First, we need to define a new type to represent the IO side-effect. This is achieved by declaring a *Java* class named *IO*. This class has one attribute named *action* which is implemented using *Java's Supplier functional interface*.

```
/**
 * The IO action which when performed will produce a IO side-effect
 * followed along with the resultant value.
 *
 * @param T The type of result this IO action produces.
 */
class IO<T> {

  /**
   * The action which when performed will produce the side effect
   * and the result.
   */
  private Supplier<T> action;

  /**
   * Creates the IO action using the action logic supplied.
   *
   * @param action The IO action logic.
   */
  public IO(Supplier<T> action) {
    this.action = action;
  }

  /**
   * Performs the IO action and returns the result.
   *
   * @return The result of the IO action obtained after
   * performing the action.
   */
  public T unwrap() {
    return this.action.get();
  }

  /**
   * Applies the transformer function to the contents of this
   * IO action to produce a larger action. The effect of this larger
   * action is same as performing each action sequentially.
   *
   * @param transformer The transformer function.
   *
   * @return The transformed IO action.
   */
  public <S> IO<S> bind(Function<T, IO<S>> transformer) {
    return transformer.apply(this.action.get());
  }
}
```

Listing 30 IO monad implemented in Java.

It is a no-args function that simply produces an output of the desired type. The IO *monad* is made generic to cater to a variety of return values. It the language does not support generics – like *Go* – the same effect can be achieved by using interfaces.

Second, we define the *unit* operation for the *monad*. In our case, the constructor of the IO *monad* becomes the *unit*. It takes in an *action* (which is a *Supplier*) which when performed will produce an output of the desired type along with the IO side-effect.

Third, we define the *unwrap* operation that enables us to explicitly perform the side-effect and unwrap the value contained in the *monad*. This is implemented as a method and when invoked it returns the execution result of the *action*.

Finally, we define the *bind* operation. This is a method that takes a *Java Function* named *transformer* as input and applies it on the result of the *monad's* own *action*. The logic of this *bind* method will vary depending upon the type of side-effect we are representing as the *monad*.

This IO *monad* (complete source code provided in Listing 30) can then be used to make impure functions like *impureSuccessor* in Listing 26 into pure functions. However, the new functions produce computations instead of values – *one-step-lazy*.

5.3 MONADIC QSTM

This section discusses the use of *monads* in the QSTM design pattern. This is inspired from Jones' discussion about the STM *monad* in *Haskell* in [7]. The UML *class diagram* in Figure 8 provides an overview of the classes and interfaces that make up the Monadic QSTM's STM implementation layer. Notice the addition of the *STMAction monad*.

The *transactional-actions* being submitted to the STM should in no way affect any resource not being maintained by the STM, so, IO actions should not be mixed in into these

transactional-actions. Moreover, when we *create*, *update*, *read*, and *delete* memory cells in the STM, internally there are several implicit side-effects taking place. If we make these implicit side-effects explicit by using the monad pattern, we can purify the *transactional-actions* to only be limited to STM specific actions.

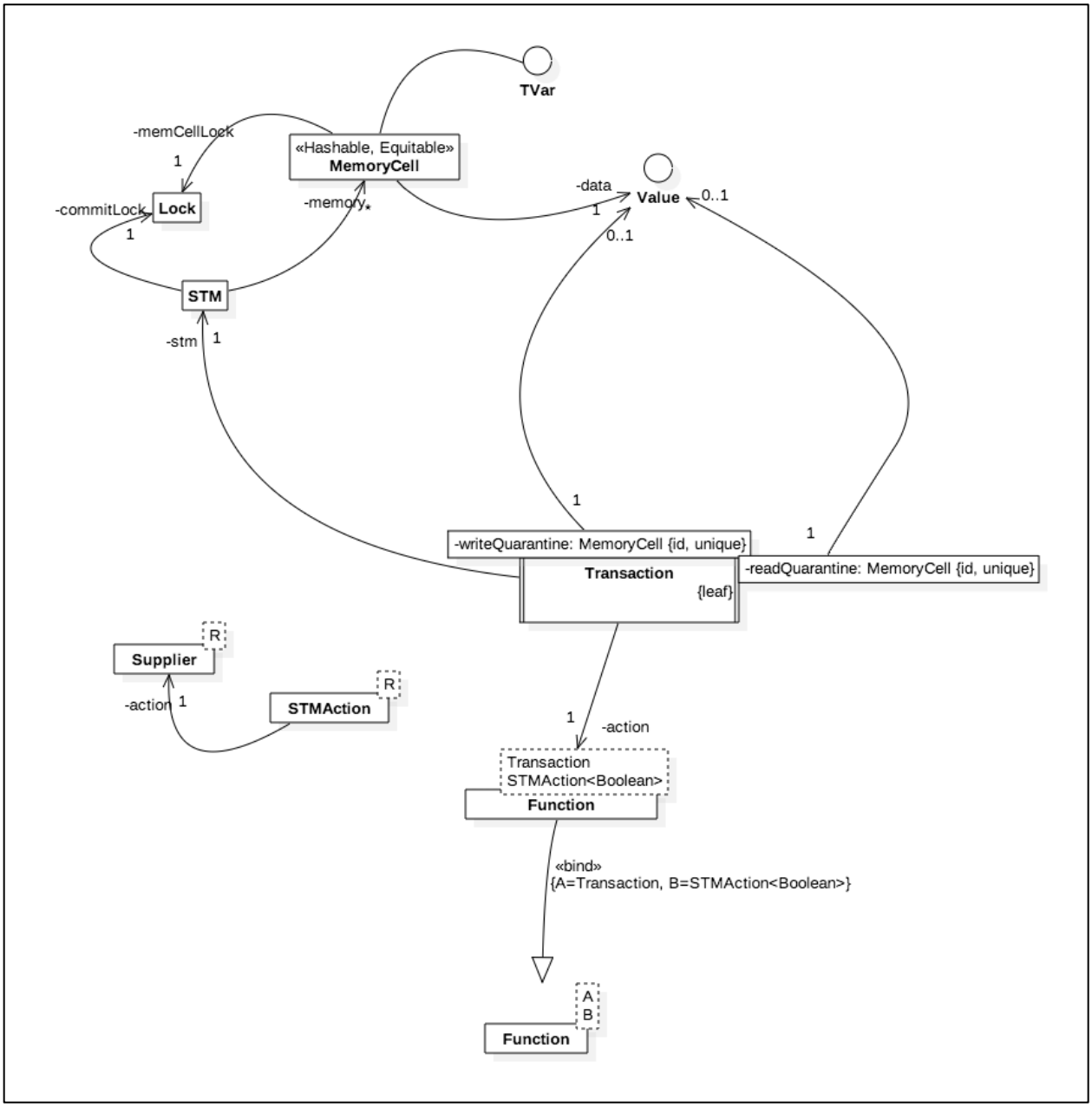

Figure 8 Monadic QSTM pattern UML overview.

The *STMAction monad* has been introduced to make the implicit side-effects of operations on the STM explicit. By using the *monad pattern* discussed in Section 5.2, we design the *STMAction monad* and the STM layer changes as shown in the UML class diagram in Figure 9.

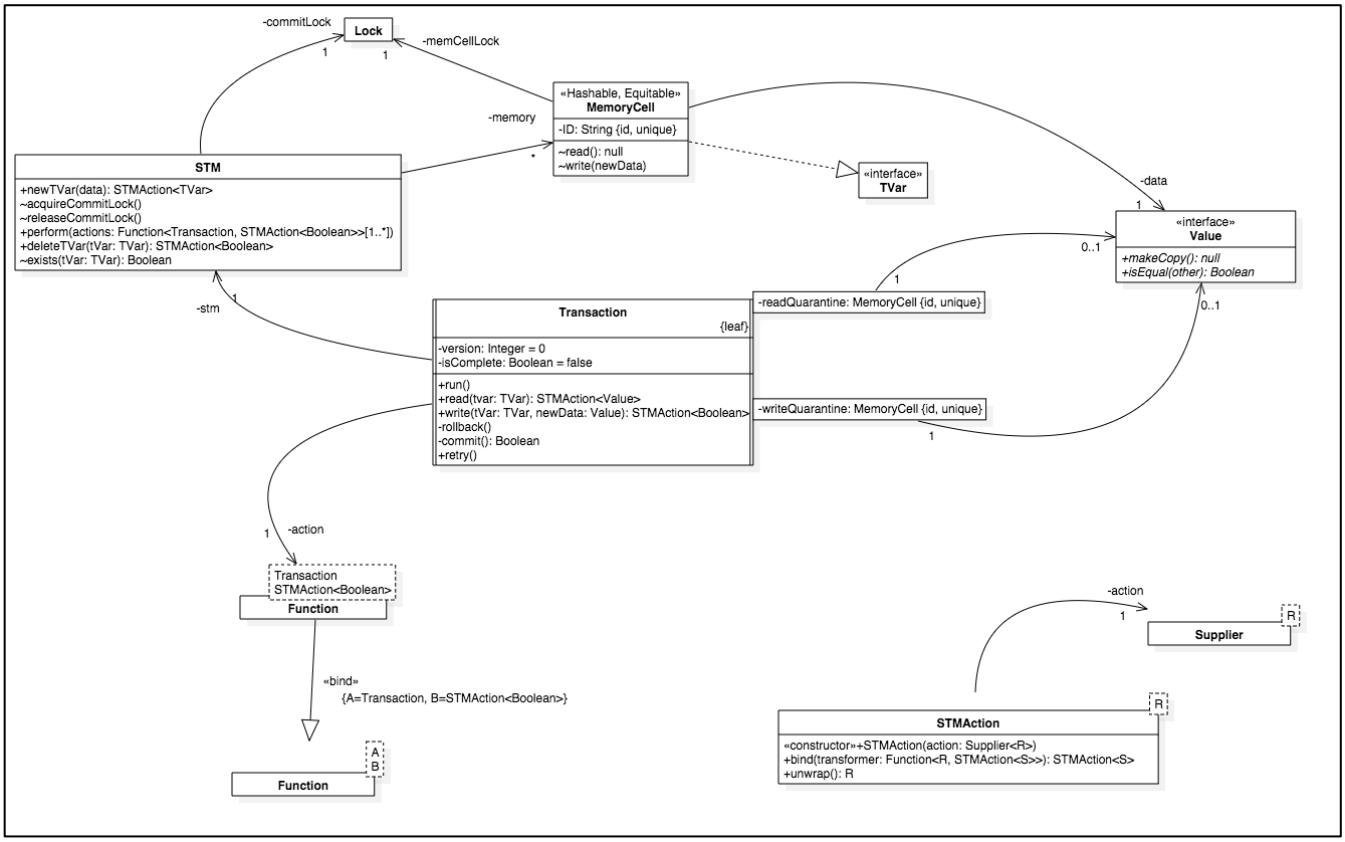

Figure 9 UML class diagram for Monadic QSTM.

We introduce the *STMAction monad* in the operations that implicitly affect the STM: *newTVar*, *deleteTVar*, *transactional read* and *write*, and *transactional-actions*.

5.3.1 STM#newTVar(Value)

The *newTVar* operation of the STM creates a new memory cell and stores the data provided in that memory cell. Creation of memory cells and adding them to the STM are implicit

side-effects that take place when this operation is performed. So, instead of returning just the *TVar* as in the non-monadic version of the QSTM pattern in Section 4.2, we return an *STMAction<TVar>*.

The *STMAction<TVar>* is a *monad* that represents an STM specific side-effect/action which when performed would create a memory cell, add it to the STM, and dump the data provided into it. Then, it will return the *transactional-variable* reference or *TVar*. The *Supplier* encapsulates the execution logic of the *STMAction*. Similar to the IO monad in Section 5.2, the *STMAction* monad also makes the operations *one-step-lazy* by wrapping them in the *Supplier* functions. The source code snippet in Listing 31 shows a possible *Java* implementation of the *newTVar* operation returning an *STMAction monad*.

```
/**
 * Makes a new transactional variable holding the provided data.
 * Internally it is a memory cell containing the data.
 *
 * @param data The data to be put into the transactional variable or memory cell.
 *
 * @return An STM action that when performed returns the transactional variable
 *    or memory cell holding the data.
 */
public STMAction<TVar> newTVar(Value data) {
  return new STMAction<>(() -> {
    MemoryCell memCell = new MemoryCell(data);
    this.memory.add(memCell);
    return memCell;
  });
}
```

Listing 31 STM#newTVar(Value) now returns an STMAction<TVar> instead of just the TVar.

5.3.2 STM#deleteTVar(TVar)

The *deleteTVar* operation on the STM now returns an *STMAction<Boolean>* instead of just *Boolean*. The implicit side-effect made explicit in this case is removal of the memory cell from the STM's *memory*. The *Java* implementation is given in Listing 32.

```
/**
 * Removes the transactional variable from the memory. The transactions trying
 * to access this deleted transactional variable need to take special care.
 * They should abort the moment they encounter this variable.
 *
 * Basically, these transactions are invalidated since they are trying to
 * operate on memory that doesn't exist anymore.
 *
 * @param tVar
 *          The transactional variable to get rid off.
 *
 * @return An STMAction which when performed will return the status
 *   of the removal operation.
 */
public STMAction<Boolean> deleteTVar(TVar tVar) {
  return new STMAction<>(() -> {
    MemoryCell memCell = (MemoryCell) tVar; // get the concrete memory cell
    return this.memory.remove(memCell);
  });
}
```

Listing 32 Monadic STM#deleteTVar(TVar) definition on Java.

5.3.3 Transaction#action

The *transactional-actions* now change their function signature from [*f* :: *Transaction* → *Boolean*] to [*f* :: *Transaction* → *STMAction<Boolean>*]. Translating the execution logic is pretty simple as we just wrap it inside a *Supplier* function making it *one-step-lazy*. Since we can now compose *transactional-actions* together using the *monadic bind pattern*, we no longer need the *executeActions* operation on the *Transaction*. The composition of *transactional-actions* build

a *thunk* which, when performed, will produce the side-effects encompassing all the individual *transactional-actions* and return the final *Boolean status*.

5.3.4 Transaction#read(TVar)

The *transactional-read* operation now returns an *STMAction<Value>* instead of just the *Value*. The implicit side-effect made explicit in this case is the action of copying data from the memory cell into the transaction's *read-quarantine*. The *Java* implementation is given in Listing 33.

```
public <T> STMAction<T> read(TVar tVar, Class<T> classz) {

  return new STMAction<>(() -> {
    try {

      if (Objects.isNull(tVar)) return null;

      Value data = null;

      if (Objects.isNull(this.readQuarantine.get((MemoryCell) tVar))) {
        data = ((MemoryCell) tVar).read();
        this.readQuarantine.put((MemoryCell) tVar, data);
      } else {
        data = this.readQuarantine.get((MemoryCell) tVar);
      }

      return classz.cast(data.makeCopy());
    } catch (Exception e) {
      logger.error(e.getMessage(), e);

      return null;
    }
  });
}
```

Listing 33 Monadic Transaction#read(TVar, Class) implementation in Java.

5.3.5 Transaction#write(TVar, Value)

The *transactional-write* operation now returns an *STMAction<Boolean>* instead of just the *Boolean status*. The implicit side-effect made explicit in this case is the action of writing the new data into the transaction's *write-quarantine*. The *Java* implementation is given in Listing 34.

```
public STMAction<Boolean> write(TVar tVar, Value newData) {

  return new STMAction<>(() -> {

    try {

      this.writeQuarantine.put((MemoryCell) tVar, newData);

      return true;
    } catch (Exception e) {

      logger.error(e.getMessage(), e);

      return false;
    }
  });
}
```

Listing 34 Monadic Transaction#write(TVar, Value) implementation in Java.

A possible *Java* implementation of the Monadic QSTM is located in [41]. The code snippet in Listing 35 shows how the *deposit* method's (from the bank account example in Listing 13 of Section 4.3) definition changes when *STMAction monad* is introduced. Although the introduction of the *STMAction monad* makes the code more robust by providing the side-effect some explicit type, without adequate language support, the code becomes verbose as shown in Listing 35.

```
public void deposit(Integer amount) {

  this.stm.perform((Transaction t) ->
      t.read(this.accountState, AccountState.class)
       .bind(as -> {
                as.deposit(amount);
                return t.write(this.accountState, as);
        })
  );

}
```

Listing 35 Account#deposit(Integer) definition in Java using monadic QSTM.

Monads are first-class citizens in *Haskell*. Therefore, it has special syntactic features developed for handling monads with ease. Unless there is similar maturity in the programming language syntax and type system, the monadic QSTM implementation will remain crude and verbose.

6 REDUX AND QSTM PATTERN

This section provides a brief overview about *Redux* [11] and compares it with our QSTM pattern.

6.1 REDUX

Redux [11] is a *state-store*. It allows the modification of the stored state only through *actions*. We define how actions affect the stored state by defining a pure function called the *reducer*. The reducer is a pure function that takes as input the current state and the incoming *action* and produces the next state (new state) as output. The new state is a brand-new object and not a mutation of the old state. Internally, *Redux* stores these transformed states in a single tree/graph.

Since the state history is being maintained in a single *state tree*, reasoning about the change in state becomes trivial. The motivation behind developing *Redux* as stated in [11] is to make state mutations predictable. For this reason, *Redux's* architecture focusses on *unidirectional data flow* like *Flux* [42]. Although *Redux* was inspired by Flux, it has some differences as mentioned in [11]: it does not have a *dispatcher* and it assumes we never mutate the data inside the *reducers*.

Redux is written in JavaScript and is *open-source*. The source code is hosted at [43]. It is mostly used on the view layer to handle the state of single page applications (SPA). It can be seen frequently used with *React* [44] library.

6.2 COMPARING REDUX AND QSTM

Although *Redux* is used in web applications, it is strictly located in the view layer. So, every client — which runs in a web-browser — is going to have its own state store which is unrelated to other clients — they do not share memory space. Moreover, the execution model is going to be sequential because JavaScript code runs in a single-thread. Nonetheless, the *Redux* state store is shared by the components/objects that make up the view of the web application's view layer: buttons, forms, labels, UI features, etc.

Although *Redux* targets an execution model that is single-threaded, it was introduced to make changes to state predictable. The JavaScript programming model makes use of asynchronicity to achieve pseudo-concurrency; mutation together with asynchronicity make programming complex [11, 43]. The main reason for introducing a state store was to maintain the unidirectional data flow and provide some form of predictability about the application state.

Our QSTM, however, is not predictable — owing to multithreading [5]. When we submit a *transactional-action*, it is not guaranteed that the state of the memory cells will get updated immediately. However, since the STM guarantees *atomicity* and *serializability*, the memory cells will be updated eventually, and their states will be *consistent* (depends upon the application's logic).

Another difference in how QSTM and *Redux* handle mutation in state is that, *Redux* does not mutate state. *Redux*, using its reducer function, transforms the old state into a new state. However, our QSTM's STM changes the state of the memory cell *in-place*. Therefore, QSTM does support mutation in state.

Although QSTM and *Redux* target different programming models — asynchronous and sequential vs. multithreaded — they converge trying to address one key issue; unpredictable mutation in state is troublesome. The *React* and *Redux* combination can be seen following the QSTM pattern. The *React* components have their identity and state separated — *props* and state [44]. The identity/*props* is immutable and the mutable state is maintained in *Redux* (the state store). Although, the state is not maintained in a STM — it is maintained in *Redux's* state tree — it is acceptable since the execution mode is sequential.

This comparison also shows the validity of the idea of isolating identity and state, storing the state in an external state store, and having the state store manage the changes in state. Having a *single source of truth* provides better control and the code is easier to write and maintain as well as reason about.

7 CONCLUSION AND FUTURE WORK

Reiterating the views of Sutter, Larus, and Lee from [4, 5], in order to write reliable, predictable, and modular concurrent programs, we require OO-like higher-level abstractions. Our QSTM pattern discussed in Section 4.2 provides such higher-level abstraction for building concurrent programs. By decoupling identity from state, we are able to tackle the nuances of shared state in the multithreaded programming model. The bank account domain example discussed in this paper demonstrates the power of the QSTM pattern.

By making implicit actions explicit, we get more control over the operations. The *monad pattern* discussed in Section 5.2 provides the tools for handling implicit actions, and the *monadic bind* discussed in Section 5.1.4 provides a way to compose together functions with side-effects. We also discussed how the monad pattern fits easily in the QSTM pattern in Section 5.3. The *monadic* QSTM is more robust because of more control over the implicit actions involved.

Although the QSTM's STM layer increases memory consumption, the increased productivity and modularity achieved can be considered a beneficial trade-off. The implementations in *Java* and *Go* also show that the QSTM pattern is versatile and can be adopted by programming languages following different programming paradigms.

QSTM's STM implementation remains in the *infrastructure layer*. So, a possible future extension could be incorporating the generic version into a programming language's standard library or the language itself. Another possibility could be adding type support for algebraic data types to programming languages like *Java* and using these types to provide a better implementation of the *monadic* QSTM. Helper utilities, meta-language processors for generating code to make objects adhere to QSTM pattern, are also possible future research areas.

REFERENCES

[1] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99-116, 1997.

[2] G. Community, "The Go programming language," 2010. [Online]. Available: https://golang.org/. [Accessed 2018].

[3] S. Lu, S. Park, E. Seo and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 329-339, 2008.

[4] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54-62, 2005.

[5] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33-42, 2006.

[6] Lightblend Inc, "Akka," Lightblend Inc, 2011. [Online]. Available: https://akka.io/. [Accessed April 2018].

[7] S. P. Jones, "Beautiful concurrency," *Beautiful Code: Leading Programmers Explain How They Think*, pp. 385-406, 1 May 2007.

[8] M. Weimerskirch, "Software Transactional Memory," 22 January 2008. [Online]. Available: https://michel.weimerskirch.net/wp-content/uploads/2008/02/software_transactional_memory.pdf. [Accessed 2018].

[9] R. Hickey, "The Clojure programming language," 2008. [Online]. Available: https://clojure.org/. [Accessed 2018].

[10] haskell.org, "Haskell Language," haskell.org, 2014. [Online]. Available: https://haskell-lang.org/. [Accessed 2018].

[11] Redux Community, "Redux," Redux Community, June 2015. [Online]. Available: http://redux.js.org/. [Accessed April 2018].

[12] G. Community, "Effective Go," 2010. [Online]. Available: https://golang.org/doc/effective_go.html. [Accessed 2018].

[13] R. G. Lavender and D. C. Schmidt, "Active object--an object behavioral pattern for concurrent programming," 1995.

[14] G. A. Agha, "Actors: A model of concurrent computation in distributed systems," MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[15] G. A. Agha, "A model of concurrent computation in distributed systems," MIT, 1986.

[16] D. G. Kafura and e. al., "ACT++ 2.0: A class library for concurrent programming in C++ using Actors," *Journal of Object-Oriented Programming*, pp. 47-56, October 1992.

[17] École Polytechnique Fédérale Lausanne (EPFL), "The scala programming language," École Polytechnique Fédérale Lausanne (EPFL), 2002. [Online]. Available: https://www.scala-lang.org/. [Accessed 2018].

[18] Ericsson AB, "Erlang programming language," 2010. [Online]. Available: http://erlang.org. [Accessed 2018].

[19] M. Herlihy, V. Luchangco and M. Moir, "A flexible framework for implementing software transactional memory," in *ACM Sigplan Notices*, 2006.

[20] M. Couceiro, P. Romano, N. Carvalho and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*, 2009.

[21] G. Korland, N. Shavit and P. Felber, "Deuce stm: Noninvasive concurrency with Java STM," in *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[22] K. E. Moore and e. al, "LogTM: log-based transactional memory," in *HPCA*, 2006.

[23] J. Pearce, "Layered architecture," [Online]. Available: http://www.cs.sjsu.edu/faculty/pearce/modules/patterns/analysis2/layers/index.htm. [Accessed 2018].

[24] Oracle, "Java thread pools," [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html. [Accessed 2018].

[25] S. Mishra, "QSTM," 2018. [Online]. Available: https://github.com/sidmishraw/mustash-stm-v2/tree/quarantined-and-fattened. [Accessed 2018].

[26] The Project Lombok Authors, "Project lombok," The Project Lombok Authors, [Online]. Available: https://projectlombok.org/. [Accessed 2018].

[27] QOS.ch, "Simple logging facade for java (SLF4J)," QOS.ch, [Online]. Available: https://www.slf4j.org/. [Accessed 2018].

[28] QOS.ch, "Logback project," QOS.ch, [Online]. Available: https://logback.qos.ch/. [Accessed 2018].

[29] Oracle, "Class UUID," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html. [Accessed 2018].

[30] Oracle, "Class ReentrantReadWriteLock," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html. [Accessed 2018].

[31] Oracle, "Interface Runnable," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html. [Accessed 2018].

[32] Oracle, "Class Hashmap," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. [Accessed 2018].

[33] Ruby community, "Ruby programming language," [Online]. Available: https://www.ruby-lang.org/en/. [Accessed 2018].

[34] Oracle, "Java collections framework," [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html. [Accessed 2018].

[35] S. Mishra, "GoSTM," 2018. [Online]. Available: https://github.com/sidmishraw/gostm/tree/quarantined. [Accessed 2018].

[36] M. McGranaghan, "Go by Example: Goroutines," [Online]. Available: https://gobyexample.com/goroutines. [Accessed 2018].

[37] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992.

[38] "Points free style," [Online]. Available: https://wiki.haskell.org/Pointfree. [Accessed 2018].

[39] S. Wlaschin, "Functional programming design patterns," [Online]. Available: https://fsharpforfunandprofit.com/fppatterns/. [Accessed 2018].

[40] Wikimedia community, "Builder pattern," [Online]. Available: https://en.wikipedia.org/wiki/Builder_pattern. [Accessed 2018].

[41] S. Mishra, "Monadic QSTM," 2018. [Online]. Available: https://github.com/sidmishraw/mustash-stm-v2/tree/monadic-qstm. [Accessed 2018].

[42] Facebook, "Flux: application architecture for building user interfaces," Facebook, [Online]. Available: https://facebook.github.io/flux/. [Accessed 2018].

[43] Redux community, "Redux repository," [Online]. Available: https://github.com/reactjs/redux. [Accessed 2018].

[44] Facebook, "React: a javascript library for building user interfaces," Facebook, [Online]. Available: https://reactjs.org/. [Accessed 2018].

[45] U. Ozgur, "Object Oriented Functional Programming or How Can You Use Classes as Redux Reducers," Medium, Feb 2017. [Online]. Available: https://medium.com/@ustunozgur/object-oriented-functional-programming-or-how-can-you-use-classes-as-redux-reducers-23462a5cae85.

[46] Wikimedia Community, "Object-oriented programming," Wikimedia, 25 September 2017. [Online]. Available: https://en.wikipedia.org/wiki/Object-oriented_programming#History. [Accessed 2018].